Graphes

D. Sarni – L. Lemarchand

Contents

Préambule

Il est communément admis que la Recherche Opérationnelle (R.O.). en tant qu'activité scientifique organisée, est née durant la seconde guerre mondiale. Des groupes de chercheurs attachés à des organismes de défense avaient alors pour tâche de donner le maximum d'efficacité à différentes "opérations" millitaires (d'où le nom de R.O.).
La recherche opérationnelle consiste en une analyse scientifique de "systèmes opérationnels" (entreprises, administrations, ...) dans lesquels des moyens humains et matériels ont été engagés dans un environnement naturel.
Les domaines d'intervention de la R.O. sont très divers (social, économique, militaire ...). En tant que science, la R.O. inter-agit avec d'autres activités scientifiques comme les mathématiques ou l'informatique, qu'elle utilise et qu'elle enrichit aussi.
Cependant un emploi sans discernement de la R.O. en tant qu'aide à la décision d'opérateurs, peut conduire dans certaines situations à des erreurs. Ces erreurs sont souvent conséquentes à un mauvais emploi de techniques issues de la R.O. ou à l'inadaptation de ces techniques par rapport à la réalité d'un problème,
L'étude d'un projet de R.O. pour être bien menée doit en général passer par les étapes suivantes:

1. **La définition des objectifs**: A ce stade, on détermine ce que le projet est supposé accomplir. Pratiquement cela consiste entre autre:
   - A tenir compte de toute hypotèse et commentaire des personnes ou organisations impliquées dans ce projet.
   - A réexaminer toute notion préconçue liée au problème que l'on traite.
   - A se mettre à la place d'un opérateur et d'examiner le projet sous cet angle.

2. **L'évolution du plan de projet**: Il faut à ce niveau, planifier l'évolution dans le temps et l'espace du projet, définir une chronologie et une répartition des différentes tâches à accomplir.

3. **La formulation du ou des problèmes rencontrés**: C'est une étape fondamentale dans la conduite du projet. Elle nécessite une information la plus complète possible afin de réunir suffisamment de données pour mieux cerner le ou les problèmes. A un niveau plus formel des supports symboliques de ces données sont utilisables (variables, constantes, relations ...).

4. **Le modèle**: Un modèle exprime une ou plusieurs relations entre les différentes variables et constantes. D'un modéle construit dépend l'efficacité d'éventuels décisions à prendre.
   Un modèle peut être décrit à l'aide d'un langage formel ou dans un langage naturel.
   Exemple.
   Supposons que l'on se trouve devant le problème suivant:

   Formulation en langage naturel. *Une entreprise conçoit et commercialise deux produits, le premier est vendu avec un bénéfice de 10 francs l'unité, le second avec un bénéfice de 20 francs l'unité. La production étant soumise à la contrainte suivante: La quantité journalière fabriquée ne peut dépasser 100 unités tous produits confondus.*
   On demande de définir un plan de production quotidien qui optimise le bénéfice de l'entreprise.
   Dans cette formulation,
   - les constantes sont 10, 20, 100,
   - les variables sont les quantités de produits fabriquées chaque jour,

- les relations doivent exprimer le bénéfice et le fait que les quantités produites sont positives et ne peuvent dépasser quotidiennement 100 unités.

Formulation dans le langage des mathématiques. On retrouvera dans cette formulation les constantes $10,\ 20,\ 100$. Mais on introduira des noms de variable: x resp. y pour représenter les quantités de produit 1 resp. 2. D'autre part on utilisera une fonction f représentant le bénéfice telle que $f(x, y) \ = \ 10 \times x + 20 \times y$. On utilisera enfin une relation d'ordre pour exprimer les contraintes de production: $x + y \leq 100,\ 0 \leq x,\ 0 \leq y$. Finalement on arrive à la formulation mathématique suivante:

$$
\begin{array}{ll}
Maximiser & f(x,\ y) \ = \ 10 \times x + 20 \times y; \\
telque & \quad x + y \leq 100; \\
 & \qquad 0 \leq x,\ 0 \leq y.
\end{array}
$$

5. **Le traitement automatique**: A ce stade, il faut:
- Développer une approche "informatique" dans les solutions au(x) problème(s) (à travers par exemple la construction d'algorithmes de résolution dont la mise en oeuvre sous forme de programmes informatiques est réalisable) et de manière plus générale dans la gestion du projet.
- Définir les spécifications des éléments intervenant dans les solutions informatiques retenues. Ces spécifications devant permettre une définition abstraite mais suffisamment précise de ces éléments.

6. **La validation du modèle**: On teste á partir de jeux de données judicieusement choisies, le modèle et les solutions retenues.

7. **L'implantation et la maintenance informatique du modèle et des solutions**.

Chapter 1

Relations, ensembles ordonnés.

1.1 Définition.

Une relation n-aire $\Re$, $(\forall n,\ n \in \mathbf{N})$, sur un ensemble E est un sous-ensemble $E_{\Re}$ du produit cartésien $E \times E \times ... \times E = E^n$.
$E_{\Re}$ est appelé graphe de la relation $\Re$.

Notations. Si $(e_1, e_2, ..., e_n) \in E_{\Re}$, *on dira que* $e_1, e_2, ..., e_n$ *sont en relation suivant* $\Re$ *et on notera* $\Re(e_1, e_2, ..., e_n)$.
Une relation n-aire peut-être totale sur un ensemble E si elle met en relation tout n-uple dans E^n, elle est partielle sinon.
Pour la suite nous ne considèrerons que des relations binaires.

1.2 Exemples.

Exemple 1. Considérons les ensembles suivants:
- X un ensemble de noms;
- Y un ensemble de dates;
- Z un ensemble d'adresses.
On peur définir sur $X \times Y \times Z$ la relations $\mathcal{R}$ telle que:
$\forall (x, y, z) \in\ X \times Y \times Z,\ \mathcal{R}(x, y, z)$ ssi l'individu dont le nom est x est né à la date y et habite à l'adresse z.

Exemple 2. Soit $X = \{x, y, z\}$ un ensemble de 3 individus participant à un tournoi. Sur X on peut défiinir les relations suivantes:
- $\Re_1(x, y)$ ssi x a rencontré y.
- $\Re_2(x, y)$ ssi x a battu y.
- $\Re_3(x, y)$ ssi x est plus agé que y.
- $\Re_4(x, y)$ ssi x a le même âge que y.

1.3 Quelques propriétés des relations binaires.

Une relation $\Re$ sur un ensemble X est

- Réflexive sur X ssi $\forall x \in X,\ \Re(x, x)$,
- Symétrique sur X ssi $\forall x, y \in X,\ \Re(x, y) \Rightarrow \Re(y, x)$,
- Antisymétrique sur X ssi $\forall x, y \in X,\ \Re(x, y)\ et\ \Re(y, x) \Rightarrow x = y$,
- Transitive sur X ssi $\forall x, y, z \in X,\ \Re(x, y)\ et\ \Re(y, z) \Rightarrow \Re(x, z)$

Retour à l'exemple 2.

- $\Re_1$ est symétrique,
- $\Re_2$ n'a aucune des propriétés définies précédemment,
- $\Re_3$ est réflexive, transitive et antisymétrique,
- $\Re_4$ est réflexive, transitive et symétrique.
  On supposera que x=y ssi x a le même âge que y.

1.4 Relations d'ordre, Relations d'équivalence.

1- Définition 1. Une relation réflexive et transitive est une relation de *préordre.*

2- Définition 2. Une relation de préordre antisymétrique est une relation *d'ordre.*

3- Définition 3. Une relation réflexive, transitive, et symétrique est une relation *d'équivalence.*

Exemples.

- La relation $\Re_3$ de l'exemple 2 est une relation d'ordre,
- $\Re_4$ est une relation d'équivalence.

4- Remarques.

1. Une relation d'ordre $\Re$ sur un ensemble X est totale ssi $\Re$ est une relation totale sur X, sinon on pourra dire que $\Re$ est une relation d'ordre partielle.
2. Un ensemble X sera dit totalement (resp: partiellement) ordonné par une relation $\Re$ ssi $\Re$ une relation d'ordre totale (resp: partielle) sur X.

5- Elements particuliers sur un ensemble ordonné. Soit X un ensemble partiellement ordonné par une relation $\Re$ et soit P une partie de cet ensemble,

- $m \in X$ est un *minorant* de P ssi $\forall x \in P\ \Re(m, x)$,
- $M \in X$ est un *majorant* de P ssi $\forall x \in P\ \Re(x, M)$,
- $n \in P$ est un élément *minimal* de P ssi $\forall x \in P\ non(\Re(x, n))$,
- $N \in P$ est un élément*maximal* de P ssi $\forall x \in P\ non(\Re(N, x))$,
- $s \in P$ est un *minimum* (*plus petit élément*) de P ssi $\forall x \in P\ \Re(m, x)$,
- $S \in P$ est un *maximum* (*plus grand élément*) de P ssi $\forall x \in P\ \Re(x, S)$,
- La *borne inférieure* de P (notée $Inf(P)$ est le plus grand des minorants de P.
- La *borne supérieure* de P (notée $Sup(P)$ est le plus petit des majorants de P.

6- Exemple (Exercice). On considère l'ensemble $X = \{1, 2, 3, 5, 10, 20, 30\}$ partiellement ordonnée par la relation $\Re_5$ telle que $\forall x, y \in X,\ \Re_5(x, y)$ ssi x divise y.
1- Monter que $\Re_5$ est une relation d'ordre partielle sur X.
2- Soit la partie $P = \{2, 5, 10\}$ de X. Préciser dans ce cas, quels sont les éléments particuliers définis précédemment.

7- Quelques opérations entre relations. Dans tout ce qui suit, $\Re$, $\Re_1$, $\Re_2$ désigneront des relations binaires sur un ensemble X. x, y désigneront des éléments quelconques dans X

7.1- Le complément. La relation $\overline{\Re}$ complémentaire de $\Re$ est par définition la relation qui a pour graphe le complémentaire du graphe de $\Re$ dans $X \times X$.

7.2- L'union "$\cup$".

$$\Re_1 \cup \Re_2(x,\ y)\ ssi\ \Re_1(x\ y)\ ou\ \Re_2(x, y)$$

7.3- L'intersection "$\cap$".

$$\Re_1 \cap \Re_2(x,\ y)\ ssi\ \Re_1(x,\ y)\ et\ \Re_2(x,\ y)$$

7.4- L'inverse "$^{-1}$".

$$\Re^{-1}(x,\ y)\ ssi\ \Re(y,\ x)$$

7.5- Le produit "$\times$".

$$\Re_1 \times \Re_2(x,\ y)\ ssi\ \exists z \in X\ tel\ que\ \Re_1(x,\ z)\ et\ \Re_2(z,\ y)$$

7.6- La fermeture transitive $^+$.

$$\Re^+ = \cup_{i \geq 1}\ \Re^i\ avec\ \Re^0 = Id_{X \times X}\ et\ \Re^i = \Re^{i-1} \times \Re$$

7.7- La fermeture réflexive transitive *.

$$\Re^* = Id_{X \times X}\ \cup\ \Re^* = \cup_{i \geq 0}\ \Re^i$$

8- Définition 4. Soit $x \in X$, la classe d'équivalence de x par rapport à une relation d'équivalence $\Re$ notée $\overline{x}$ est par définition l'ensemble $\{x' \in X,\ \Re(x,\ x')\}$.

Propriétés. Dans tout ce qui suit on supposera définie une relation d'équivalence $\Re$ sur un ensemble X. x désignera un élément de X.

1. $x \in \overline{x}$,
2. $\forall x,\ x' \in X\ , \Re(x,\ x') \Rightarrow \overline{x} = \overline{x'}$,
3. $\forall x,\ x' \in X\ , \overline{x} \cap \overline{x'} \neq \emptyset \Rightarrow \overline{x} = \overline{x'}$.

9- Définition 5. Soit $\Re$ une relation d'équivalence sur un ensemble X, *L'ensemble quotient* de X par $\Re$ est par définition l'ensemble $X/\Re = \{\overline{x},\ x \in X\}$.

Propriétés.

$$\cup_{\overline{x} \in X/\Re} = X$$

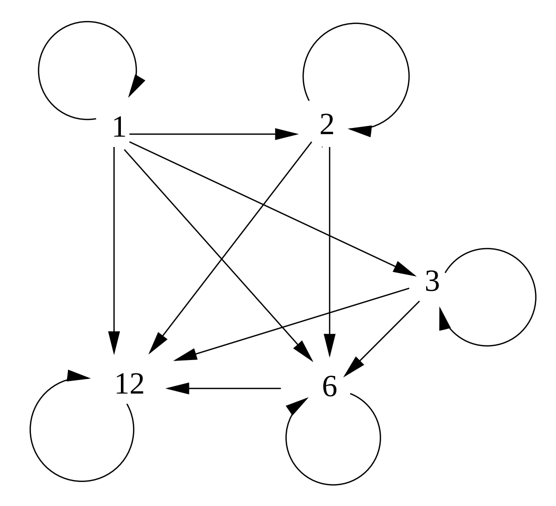
Comme les classes d'équivalence distinctes ont des intersections vides, on dira que $X/\Re$ définit une partition de X.

1.5 Représentations d'une relation.

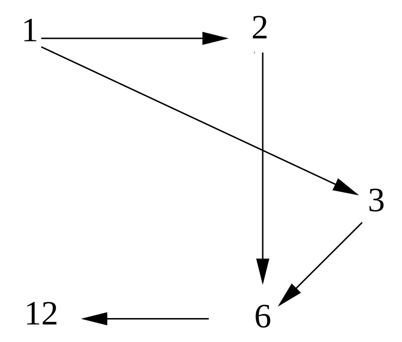
Soit $\mathcal{R}$, une relation sur un ensemble X, on pourra représenter le fait que deux éléments a, b de X sont en relation par un arc de a vers b: $a \longrightarrow b$. si $a,\ b$ et $b,\ a$ sont en relation, on aura une arête non-orientée $a - - - b$.
Une relation sera donc représentée par un ensemble d'arcs (ou d'arêtes non-orientées) pouvant se succéder.
Exemple.
La relation $\mathcal{D}$ définie sur $X = \{1,\ 2,\ 3,\ 6,\ 12\}$ telle que:

$$\forall x,\ y\ \in X,\ \mathcal{D}(x,\ y)\ ssi\ x\ divise\ y$$

peut être représentée par:



Ce type de représentation est appelée représentation sagitale. On remarquera que 1 est origine d'arcs ayant pour autre extrémité chacun des autres éléments et que d'autre part 12 est une extrémité finale d'arcs dont l'autre extrémité est un autre élément quelconque de X.
Les arcs peuvent être étiquetté. Dans l'exemple précédent, une étiquette d'un arc pourrait être un entier dont le produit avec l'entier à l'origine de l'arc donnerait l'entier à l'extrémité finale de l'arc.
Dans le cas d'une relation d'ordre, si l'on supprime les boucles (dues à la réflexivité) et les arcs dus à la transitivité, on obtient un diagramme de Hasse.
Le diagramme de Hasse de l'exemple précédent est



Ces deux représentations sont des "graphes" particuliers.

Exercice.
Représentation d'un ordonnancement. Etude d'un cas.

Chapter 2

Eléments de base

2.1 Graphe orienté – Définitions

- Un *graphe orienté* G consiste en la donnée d'un couple d'ensembles (X, U).
- X est appelé ensemble des *sommets* du graphe G.
- U est appelé ensemble des *arcs* du graphe G.
- Tout arc $u \in U$ est un couple ordonné de sommets (i, j). i est l'extrémité *initiale* de u. j est l'extrémité *finale* de u. i est un *prédecesseur* de j et j est un sommet *successeur* de i. Une boucle est un arc dont les extrémités coïncident.

- Si $|X| = n$, on dira que G est d'ordre n.
- Un p-graphe est un graphe G tel que :

  $\forall i, j \in X$, l'ensemble des arcs (i, j) a un cardinal fini inférieur ou égal à p.

2.2 Représentation graphique

Graphiquement, chaque sommet d'un graphe G est représenté par un point ou un cercle, et chaque arc $u = (i, j)$ par une flèche joignant le point i au point j (orientée vers j) (figure 2.1).

figure=a.ps,width=7cm,height=4cm

Figure 2.1: Un 2-graphe avec une boucle

2.3 Graphe non orienté

Etant donné un ensemble X de sommets, une arête, de sommets i et j, est un couple non ordonné (i, j) d'éléments de X. Un *graphe non orienté* G' consiste en la donnée d'un ensemble X de sommets et d'un ensemble U' d'arêtes. Graphiquement, une arête est représentée par un segment.

figure=b.ps,width=7cm,height=4cm

Figure 2.2: Un graphe non orienté avec une boucle

- Un *multigraphe* est un graphe tel qu'il peut exister plus d'une arête entre deux sommets.
- un graphe sans boucle tel que deux sommets sont joints par au plus une arête est un graphe *simple*.

remarque A un graphe orienté $G = (X, U)$, on peut toujours associer un graphe non orienté G' en considérant U comme un ensemble d'arêtes. C'est le cas des graphes des figures 2.1 et 2.2.

Dans la suite, on dira graphe pour graphe orienté, sauf indication contraire.

2.4 Terminologie

- *Arcs (arêtes) adjacents (es)* :

Deux arcs (arêtes) sont adjacents (es) si ils ont au moins une extrémité commune.

exemple Dans le graphe de la figure 2.1, u_2 et u_3 sont adjacents (au sommet 2).

- *Degré, demi-degré* :

Le demi-degré *extérieur* d'un sommet i, noté $d^+(i)$ est le nombre d'arcs ayant i comme extrémité initiale.

Le demi-degré *intérieur* d'un sommet i, noté $d^-(i)$ est le nombre d'arcs ayant i comme extrémité finale (ou terminale).

le degré d'un sommet i, noté $d(i)$ est la somme des ses demi-degrés intérieur et extérieur.

exemple Dans le graphe de la figure 2.1, $d^+(2) = 2$, $d^-(2) = 1$, et $d(2) = 2 + 1 = 3$.

- *Arcs (arêtes) incidents (es). Cocycles* :

Soit $A \in X$. A un est sous-ensemble de sommets de X. On définit les ensembles suivants :

$\Omega^+(A) = \{u \in U | u = (i, j), i \in A, j \notin A\}$

$\Omega^-(A) = \{u \in U | u = (i, j), i \notin A, j \in A\}$

$\Omega(A) = \{\Omega^+(A) \bigcup \Omega^-(A)\}$

$\Omega^+(A)$ est l'ensemble des arcs partant de A et aboutissant en dehors de A. $\Omega^-(A)$ est l'ensemble des arcs partant de l'extérieur de A pour aboutir dans A. $\Omega(A)$ est un *cocycle* du graphe.

exemple Dans le graphe de la figure 2.1, avec $A = \{2, 3\}$, on a :
$\Omega^+(A) = \{u_2, u_4, u_6, u_7\}$
$\Omega^-(A) = \{u_1\}$
$\Omega(A) = \{u_1, u_2, u_4, u_6, u_7\}$

- *Graphe (anti)symtrique* :

Un graphe G est dit symtrique ssi quels que soient les sommets i et j le nombre d'arcs (i, j) est gal au nombre d'arcs (j, i). G est antisymtrique ssi $(i, j) \in U \Rightarrow (j, i) \notin U$.

exemple Le graphe de la figure 2.1 n'est pas symtrique car, par exemple $(2, 3) \in U$ et $(3, 2) \notin U$. Il n'est pas non plus antisymtrique car $(1, 2) \in U$ et $(2, 1) \in U$.

- *Graphe complet, clique* :

Un graphe $G = (X, U)$ est complet ssi deux sommets quelconques sont joints par au moins un arc.

Un sous-ensemble C de X est une clique ssi deux sommets quelconques sont joints par une arête.

figure=c.ps,width=6cm,height=3.5cm

Figure 2.3: Un graphe orienté contenant des cliques

exemple Le graphe de la figure 2.3 n'est pas complet car aucun arc ne joint le sommet 4 aux autres. Dans le graphe non orient associ ce graphe, le sous-ensemble $\{1, 2, 3\}$ est une clique. Par contre, le sous-ensemble $\{1, 2, 4\}$ n'en est pas une.

- *Sous-graphe* :

Soit $G = (X, U)$, $G' = (X', U')$ deux graphes. On dira que G' est un sous-graphe de G ssi X' est inclus dans X et U' est inclus dans U.

exemple Le graphe de la figure 2.4 est un sous graphe de celui de la figure 2.3.

figure=d.ps,width=4.5cm,height=3.5cm

Figure 2.4: Un sous-graphe du graphe de la figure préceédente

- *Graphe bipartite* :

Un graphe $G = (X, U)$ est bipartite ssi il existe deux parties distinctes (ensembles de sommets) X_1 et X_2 recouvrant X telles que

$$\forall (i, j) \in U, \quad i \in X_1 \Rightarrow j \in X_2 \qquad et \qquad i \in X_2 \Rightarrow j \in X_1$$

exemple Le graphe de la figure 2.5 est un graphe bipartite, avec $X_1 = \{1, 2\}$ et $X_2 = \{3, 4, 5\}$.

figure=e.ps,width=4cm,height=3cm

Figure 2.5: Un graphe bipartite

2.5 Implantation d'un graphe

L'implantation d'un graphe peut se faire par le biais de différents types de matrices associées au graphe.

2.6 Matrice d'incidence *sommets-arcs*

Une matrice d'incidence sommets-arcs d'un graphe $G = (X, U)$ avec $|X| = n$ et $|U| = m$ est une $n \times m$ matrice A_{iu} telle que :

- $A_{iu} = +1$ et $A_{ju} = -1$ si (i, j) est un $u^{ème}$ arc dans G.
- $A_{iu} = 0$ sinon.

exemple La matrice de la figure 2.6(a) indique que l'on a

- un graphe avec

  4 sommets (4 lignes)
  5 arcs (5 colonnes)

- le premier arc (1^{ière} colonne) d'extrémité initiale 1 et finale 2 ($A_{11} = 1, A_{21} = -1$).
- le deuxième arc (2ème colonne) d'extrémité initiale 1 et finale 3 ($A_{12} = 1, A_{32} = -1$).

...

On obtient ainsi le graphe de la figure 2.6(b).

$$\begin{bmatrix} +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 \\ 0 & -1 & +1 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 \end{bmatrix}$$

figure=f.ps,width=4cm,height=3cm

Figure 2.6: (a) Une matrice d'incidence sommets-arcs (b) le graphe correspondant

2.7 Matrice d'incidence *sommets-arêtes*

Une matrice d'incidence sommets-arêtes d'un graphe non orienté $G = (X, U)$ avec $|X| = n$et $|U| = m$ est une $n \times m$ matrice A_{iu} telle que :

- $A_{iu} = 1$ si (i, j) est une $u^{ème}$ arête dans G.
- $A_{iu} = 0$ sinon.

exemple Le graphe non-orienté de la figure 2.7(b) a la matrice d'incidence somets-arêtes de la figure 2.7(a).

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

figure=g.ps,width=4cm,height=3cm

Figure 2.7: (a) Une matrice d'incidence sommets-arêtes (b) le graphe non orienté correspondant

2.8 Matrice d'adjacence

Soit $G = (X, U)$ un *1-graphe* comportant au plus une boucle par sommet, avec $|X| = n$. La matrice d'adjacence associée à G est une matrice $n \times n$ A_{ij} telle que

- $A_{ij} = 1$ si $(i, j) \in U$.
- $A_{ij} = 0$ sinon.

Dans le cas non orienté, la matrice d'adjacence A_{ij} d'un graphe simple est telle que :

- $A_{ij} = A_{ji} = 1$ si (i, j) est une arête du graphe.
- $A_{ij} = 0$ sinon.

exemple Le graphe non orienté de la figure 2.8(b) a la matrice d'adjacence de la figure 2.8(a).

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

figure=h.ps,width=4cm,height=3cm

Figure 2.8: (a) Une matrice d'adjacence (b) le 1-graphe correspondant

Chapter 3

Cheminement et connexité

3.1 Chaines et cycles

- *chaine, chaine élémentaire* :
  - Une chaine de longueur q (de cardinalité q) est une séquence de q arcs $L = \{u_1, u_2, ..., u_q\}$ telle que chaque arc u_r de la séquence $(2 \leq r \leq q-1)$ ait une extrémité commune avec l'arc u_{r-1} $(u_r \neq u_{r-1})$ et l'autre extrémité avec u_{r+1} $(u_r \neq u_{r+1})$.

    L'extrémité i de u_1 non adjacente à u_2 et l'extrémité j de u_q non adjacente à u_{q-1} sont appelées les extrémités de la chaine L. On dit aussi que L joint les sommets i et j.
  - Une chaine élémentaire est une chaine telle que en la parcourant on ne rencontre pas deux fois le même sommet. De manière équivalente, on peut définir une chaine élémentaire comme une chaine dont tous les sommets sont de degré 2 au plus.

Dans le cas d'un graphe simple, une chaine de longueur q est parfaitement définie par la succession des sommets qu'elle rencontre.

figure=i.ps,width=7cm,height=4cm

Figure 3.1: Une chaine (arcs en **gras**) dans un graphe

exemple Dans le graphe de la figure 3.1, $L = \{u_2, u_5, u_6, u_4\}$ est une chaine reliant les sommets 2 et 3. Cette chaine est élémentaire. L pourrait être décrite à partir de la succession de sommets suivante : 2, 1, 4, 5, 3.

- *cycle, cycle élémentaire* :
  - Un cycle est une chaine dont les extrémités coincident.
  - Un cycle élémentaire est un cycle tel que en le parcourant on ne rencontre pas deux fois le même sommet, sauf le sommet choisi comme origine du parcours.

exemple Dans le graphe de la figure 3.1, $C = \{u_1, u_3, u_6, u_5\}$ est un cycle élémentaire.

3.2 Chemins et circuits

- *chemin, chemin élémentaire* :
  - Un chemin de longueur q (de cardinalité q) dans un graphe G est une séquence de q arcs $P = \{u_1, u_2, ..., u_q\}$ telle que :

    $$u_1 = (i_0, i_1), \quad u_2 = (i_1, i_2), ..., u_q = (i_{q-1}, i_q)$$

$i_0, i_1, i_2, ..., i_{q-1}, i_q$ étant des sommets du graphe G.

Autrement dit, un chemin est une chaine dont tous les arcs sont orientés dans le même sens. i_0 est l'extrémité *initiale* de P et i_q est l'extrémité *finale* de P.

- Un chemin est élémentaire si, en le parcourant, on ne rencontre pas deux fois le même sommet.
- Dans le cas d'un 1-graphe, un chemin peut être décrit de façon équivalente par la succession ordonnée des sommets qu'il rencontre.

exemple Dans le graphe de la figure 3.1, $P = \{u_1, u_3, u_4, u_7\}$ est un chemin allant du sommet 1 au sommet 4. On peut le représenter par la séquence 1, 2, 5, 3, 4 de sommets du graphe.

- *circuit, circuit élémentaire* :
  - Un circuit (élémentaire) est un chemin (élémentaire) dont les extrémités coincident.

3.3 Connexité, nombre de connexité

- *composante connexe* :
  - Un graphe $G = (X, U)$ est dit connexe si, pour tout couple de sommets i et j de X, il existe une chaine joignant i et j. La relation

$$i \, \mathcal{R} \mid \int\!\int\rangle \quad \begin{cases} i = j \\ ou \\ i \neq j \quad et\ il\ existe\ une\ chaine\ joignant\ i\ \grave{a}\ j. \end{cases}$$

est une relation d'équivalence sur X

  - Les classes d'équivalence induites sur X par cette relation forment une partition de X. Soit $X_1, X_2, ..., X_p$ cette partition. Les sous-graphes $G_1, G_2, ..., G_p$ engendrés par les sous-ensembles $X_1, X_2, ..., X_p$ sont appelés composantes connexes de G.

- *nombre de connexité, graphe connexe*
  - Le nombre p de composantes connexes distinctes est le nombre de connexité du graphe.
  - un graphe est dit connexe si son nombre de connexité $p = 1$.
  - chaque composante connexe est un graphe connexe.

figure=j.ps,width=9cm,height=4cm

Figure 3.2: un graphe avec 3 composantes connexes

exemple La figure 3.2 représente un graphe ayant trois composantes connexes :

$G_1 = (\{1, 2, 4\}, \{u_1, u_2, u_4\})$

$G_2 = (\{5, 7\}, \{u_5\})$

$G_3 = (\{3, 8, 9\}, \{u_3, u_6, u_7\})$

La vérification de la connexité d'un graphe ou la détermination de ses composantes connexes est un des problèmes fondamentaux de la théorie des graphes.

3.4 Graphes et composantes fortement connexes

- *graphe fortement connexe*

  - Un graphe $G = (X, U)$ est dit fortement connexe si, pour tout couple de sommets i et j (dans cet ordre) de X, il existe un chemin de i à j. La relation

$$i\ \mathcal{R}' \mid\ \int\int\rangle\ \begin{cases} i = j \\ ou \\ i \neq j \quad et\ il\ existe\ un\ chemin\ joignant\ i\ à\ j. \end{cases}$$

    est une relation d'équivalence sur X.

  - Les sous-graphes $G_1, G_2, ..., G_p$ engendrés par ses classes d'équivalence $Y_1, Y_2, ..., Y_p$ sont appelés composantes fortement connexes de G.

  - Le nombre p de composantes fortement connexes distinctes est le nombre de connexités fortes du graphe.

  - Un graphe est dit fortement connexe ssi il n'a que une seule composante fortement connexe.

exemple Le graphe de la figure 3.2 a une seule composante fortement connexe de taille supérieure à 1 : $Y_1 = (\{1, 2\}, \{u_1, u_2\})$.

Chapter 4

Implantation sous forme de liste d'un graphe

Nous avons vu précédemment qu'il est possible de décrire un graphe à partir de deux familles de matrices :

La matrice d'adjacence ou ses dérivées,

La matrice d'incidence ou ses dérivées.

Dans tous les cas, on se rend compte que les matrices considérées contiennent beaucoup de zéros et donc que un stockage tel quel de ces tableaux en mémoire n'est pas judicieux. C'est pour cela que l'on utilise souvent une représentation de ces matrices sous forme de listes.

4.1 A partir de la matrice d'adjacence

On rappelle qu'une matrice d'adjacence sert à décrire soit des 1-graphes (orientés) soit des graphes simples (non orientés).

Une telle représentation peut être mise en oeuvre par des *listes d'adjacence*. Pour un graphe $G = (X, U)$, avec $|X| = N$ et $|U| = M$, on utilise deux tableaux `A[]` et `B[]` de dimensions respectives `N+1` et `M` (cas orienté) ou `2M` (cas non orienté).

Pour chaque sommet i, la liste des successeurs de i est contenue dans le tableau `B[]` à partir de la case `A[i]`. On voit donc que l'ensemble des informations relatives au sommet i est stocké entre les cases `A[i]` et `A[i+1] - 1` du tableau `B[]`. On a les égalités suivantes :

$d^+(i) =$ `A[i+1] - A[i]` (cas orienté).

$d^(i) =$ `A[i+1] - A[i]` (cas non orienté).

`A[i]` $= \sum_{j=1}^{i-1} d^+(j) \quad +1$

figure=k.ps,width=9cm,height=4cm

Figure 4.1: (a) un 1-graphe (b) liste d'adjacence associée

exemple

- la figure 4.1(b) représente les listes d'adjacence d'un 1-graphe 4.1(a), avec `N = 3` et `M = 6`.
- la figure 4.2(b) représente les listes d'adjacence d'un graphe simple 4.2(a), avec `N = 4` et `M = 5`.

remarque Lorsque le graphe est pondéré, c'est à dire que l'on associe une valeur numérique à chaque arête, on peut stocker les poids dans une liste de même taille que `B[]`.

figure=l.ps,width=9cm,height=4cm

Figure 4.2: (a) un graphe simple (b) liste d'adjacence associée

4.2 A partir d'une matrice d'incidence

Deux représentations sont essentiellement utilisées pour représenter un graphe $G = (X, U)$, avec $|U| = \mathtt{M}$.

4.2.1 Liste des arêtes

Une première méthode consiste à définir deux tableau notés `EI[]` et `EF[]` (Extrémité Initiale, Finale) de dimension `M` donnant pour chaque arc ou arête le numéro des extrémités. Dans le cas non orienté, on ne tient pas compte des qualificatifs *initial* et *terminal*.

exemple La figure 4.3(b) représente les listes d'arêtes du graphe de la figure 4.3(a).

figure=m.ps,width=10cm,height=4cm

Figure 4.3: (a) un graphe (b) liste d'arêtes associée

remarques

- Cette représentation permet de décrire les p-graphes ou des multigraphes comportant des boucles.
- Lorsque le graphe est pondéré, on ajoute une liste `P` de dimension `M` en correspondance biunivoque avec `EI[]` et `EF[]` et contenant les poids.

4.2.2 Liste des cocycles $\Omega^+(i)$ ou $\Omega^-(i)$

On établit pour chaque sommet i du graphe

- La liste $\Omega^+(i)$ des arcs ayant i pour origine (cas orienté).
- La liste $\Omega^(i)$ des arcs ayant i pour origine (cas non orienté).

Deux tableaux `LP[]` et `LA[]` (Liste des Arcs ou Arêtes) sont utilisés. Les informations relatives à un sommet i sont contenues dans `LA[]` entre les cases `LP[i]` et `LP[i+1] -1`.

Lorsque l'on utilise ce type de représentation, il est généralement nécessaire d'indiquer dans un autre tableau `LS[]`, de même taille que `LA[]`, le numéro de l'autre extrémité correspondant à i.

`LS[]` n'est pas nécessaire dans le cas où `EI[]` et `EF[]` sont en mémoire. De plus, la dimension de `LP[]` est égale à `N+1` et celle de `LA[]` et `LS[]` à `M`.

figure=n.ps,width=10cm,height=4cm

Figure 4.4: (a) un graphe (b) liste de cocycles associée

exemple Pour le graphe de la figure 4.3(a), représenté à nouveau en 4.4(b), les listes de cocycles correspondantes sont représentées sur la figure 4.4(b).

Chapter 5

Recherche dans un graphe

5.1 Objectifs de cette recherche

Etant donné un graphe $G = (X, U)$, nous allons définir un algorithme répondant aux problèmes suivants :

- Un sommet j est-il accessible par un chemin à partir d'un autre sommet i ?
- Quel est l'ensemble de tous les sommets accessibles à partir d'un sommet i donné ?

5.2 Principe du parcours en profondeur d'un graphe

Le parcours en profondeur d'un graphe est similaire au parcours en préordre d'un arbre.

Au cours de l'exploration, un sommet pourra se trouver dans deux états : `ouvert` si il n'a pas encore été visité, `fermé` sinon.

L'exploration se fait à partir d'un sommet S. La description formelle de l'algorithme est la suivante : *Si S est ouvert, on le visite puis on effectue un parcours en profondeur de tous ses sommets adjacents ouverts.*

Pour effectuer le parcours, on associe à chaque sommet un drapeau que l'on positionne lorsque l'on visite le sommet, pour le faire passer de `ouvert` à `fermé`.

exemple Le tableau de la figure 5.1(b) représente les différentes étapes de l'algorithme pour le parcours en profondeur du graphe de la figure 5.1(a). Les sommets fermés sont en gras. A chaque étape le sommet sélectioné parmi les sommets adjacents du sommet courant est souligné.

Lors des étapes 4, 5, 7, 8, 9, aucun sommet n'est sélectionné car il n'existe aucun sommet ouvert dans la liste d'adjacence du sommet courant. Cela signifie que l'on ne peut espérer de nouvelles possibilités de cheminement à partir de ce sommet.

Dans ce cas, on effectue un retour en arrière (*backtracking*) vers l'ancêtre du sommet courant (celui à partir duquel le sommet courant a été sélectionné). On relance alors l'algorithme (étapes 1 et 6 dans l'exemple). La recherche est terminée lorsque l'on revient au sommet initial, et que tous ses sommets adjacents sont fermés (c'est le seul sommet ne possédant pas d'ancêtre) (étape 9 dans l'exemple).

remarque Le choix d'un sommet courant parmi les sommets adjacents ouverts d'un sommet est arbitraire. Des choix différents peuvent mener à des ordres de visite différents pour les sommets.

exemple Par exemple, si à l'étape 2 de la figure 5.1(b) le sommet e avait été choisi à la place de c, on aurait obtenu la séquence de visite a, b, e, c, d, au lieu de a, b, c, e, d.

Le parcours d'un graphe connexe comme celui de la figure 5.1(a) partitionne implicitement l'ensemble des arcs du graphe en deux ensembles : ceux qui ne sont pas utilisés par la recherche et ceux qui sont utilisés (assortis d'une * sur la figure). L'arbre correspondant à ce second ensemble est appelé *arbre de*

figure=o.ps,width=4cm,height=4cm

#	Ancêtres et sommet courant	sommets adjacents	Commentaire
1	**a**	b	a est le sommet courant de départ. on choisit b, seul sommet adjacent de a.
2	**a → b**	c e d	on choisit c parmi les sommets adjacents de b (mais e ou c auraient pu être choisis).
3	**a → b → c**	**b** e	e est le seul sommet adjacent à c ouvert. il est choisi.
4	**a → b → c → e**	**c**	e ne possède aucun adjacent ouvert. on remonte à son ancêtre c.
5	**a → b → c**	**b e**	idem pour c : b et e sont fermés.
6	**a → b**	**c e** d	d est le seul sommet adjacent à b encore ouvert.
7	**a → b → d**	**a**	on remonte à l'ancêtre b de d, a étant fermé.
8	**a → b**	**c e d**	on remonte de b vers a.
9	**a**	**b**	a n'a pas d'ancêtre (sommet initial) et ses adjacents (b) sont tous fermés. fin.

Figure 5.1: (a) Un graphe (b) les étapes d'un parcours en profondeur à partir de a

recouvrement. Un tel arbre comprend tous les sommets de la composante connexe visité. On notera que plusieurs arbres de recouvrement peuvent exister pour un même graphe, correspondant aux différents parcours possibles.

5.3 L'algorithme de parcours en profondeur

On définit les deux fonctions suivantes :

Visiter(*Sommet S*; *Graphe G*) ( ) qui ferme un sommet S et permet d'effectuer une opération sur ce sommet, comme par exemple l'affichage des informations associées à S.

Ouvert(*Sommet S*; *Graphe G*) (*booléen*) *qui retourne* VRAI *si S est ouvert et* FAUX *si S est fermé.*

L'algorithme de parcours d'un graphe G à partir d'un de ses sommets V est alors le suivant :

```
Profondeur( Graphe G; Sommet V) ( )
    Sommet S
    Liste li_adj

    Si Ouvert( V, G ) = VRAI
        Visiter( V, G )
        li_adj ← ListSadj( V, G )
        Pour chaque S dans li_adj
            Si Ouvert( S, G ) = VRAI
                Profondeur( G, S )
```

Exercice Ecrire une version itérative de Profondeur(). On pourra utiliser une pile pour stocker le sommet courant et sa liste d'ancêtres. Cela revient à simuler les différents appels récursifs.

5.4 Parcours en largeur d'un graphe

Le parcours en largeur d'un graphe est similaire au parcours en largeur d'un arbre. On utilise les mêmes notions de sommets ouverts ou fermés que pour le parcours en profondeur. La description formelle d'un parcours en largeur d'un graphe est la suivante :
On visite le premier sommet S puis tous ses sommets adjacents ouverts. Pour chacun de ceux-ci, on examine leurs sommets adjacents ouverts, ...

Les chiffres de la figure 5.2 représentent l'ordre de visite des sommets lors du parcours en largeur du graphe de la figure 5.1(a). L'arbre de recouvrement (arcs marqués d'une *) associé est également représenté.

figure=p.ps,width=5cm,height=4cm

Figure 5.2: un graphe et son arbre de recouvrement (recherche en largeur) associé

Pour implanter l'algorithme, on utilise une pile dans laquelle on place les sommets dont la liste d'adjacence reste à explorer.

```
Largeur( Graphe G; Sommet V ) ( )
    Liste[ Sommet ] li_adj
    File[ Sommet ] pile
    Sommet S

    CréerFile( pile )
    Mettre( V, pile )
    Visiter( V, G )

    tantque Vide( pile ) = FAUX
        Retirer( S, pile )
        li_adj ← ListSadj( S, G )
        Pour chaque S dans li_adj
            Si Ouvert( S, G ) = VRAI
                Visiter( S, G )
                Mettre( S, pile )
```

5.5 Détermination des composantes connexes

Pour déterminer les composantes connexes d'un graphe, on procède comme suit :

- On choisit un sommet S_0 de G et on applique l'algorithme de recherche en profondeur décrit précédemment à partir de S_0.

- On associe à S_0 et à chacun des sommets visités lors de l'exploration un même numéro de composante connexe noté $NC(S_0)$.

- Après cette recherche, deux cas peuvent se présenter :

  1 Tous les sommets du graphe ont été atteints. Dans ce cas, le graphe G est connexe (une seule composante connexe).

  2 Certains des sommets n'ont pas été visités. Dans ce cas, on relance l'algorithme à partir de l'un d'entre eux, en incrémentant le numéro de composante connexe.

  On continue ainsi tant que il reste des sommets non explorés.

La figure 5.3 illustre l'algorithme. A chaque étape d'une recherche en profondeur, la composante connexe d'un sommet est déterminé. Les sommets initiaux des deux recherches effectuées sont 1 et 4.

figure=q.ps,width=6cm,height=4cm

Figure 5.3: Détermination des composantes connexes d'un graphe

Implántation de l'algorithme On modifie la fonction Visiter() de l'algorithme Profondeur() de façon à ce qu'elle marque les sommets visités avec un numéro de composante connexe. La fonction de parcours s'écrira alors :

```
Profondeur2( Graphe G; Sommet V; entier NC)
    ⋮
    Si ...
        Visiter( V, G, NC )
        ⋮
                Profondeur2( G, S, NC )
```

L'algorithme de détermination des composantes connexes est alors le suivant :

```
Connexité( Graphe G ) ( Liste[ Sommet ] )
    Sommet S
    Liste[ Sommet ] L

    NC ← 0
    L ← ListSom( G )

    Pour chaque S dans L
        Si Ouvert( S, G ) = VRAI
            NC ← NC + 1
            Profondeur2( G, S, NC )

    retourner L
```

L contient la liste des sommets, marqués par leur numéro de composante connexe.

Chapter 6

Le Type Abstrait de Donnée graphe.

6.1 Définition abstraite de procédures ou de types.

Avant d'écrire des programmes, il s'avère souvent nécessaire de définir de façon abstraite, une ou plusieurs procédures, un ou plusieurs nouveaux types que ces programmes vont implémenter. Il s'agira donc pour nous de donner des règles permettant d'avoir des définitions précises. Ces règles devront en outre tenir compte de la mise en oeuvre des entités qu'elles définissent.

6.1.1 Définition abstraite d'une procédure.

Une procédure est une application définie sur un ensemble de données dites "d'entrée" appelées arguments (cet ensemble peut être vide). Cette application peut retourner un résultat appelé donnée de sortie (la procédure est alors appelée fonction), ou bien réaliser une action.
Par exemple la procédure OCC1 est une fonction, la procédure SUPP n'est pas une fonction.
Pour exister en tant qu'entité abstraite, une procédure peut être spécifiée. Les normes de spécification utilisées doivent permettre:
- L'identification de la procédure,
- de préciser son mode d'utilisation,
- de préciser ses effets,
- de préciser les données qu'elle modifient.
Pratiquement, voiçi les règles de spécification de procédures abstraites que nous retiendrons:

1. Préciser dans un en-tête: le nom de la procédure précédé du type du résultat retourné (si la procédure est une fonction) et suivi de la liste des arguments et de leur type. A travers les exemples qui suivent nous décrivons la présentation d'un en-tête:
   - (*booleen*) OCC (*tableau[entier]* A; *entier* e),
   - SUPP (*tableau[entier]* A; *entier* e).
   - (*liste*) créer.

2. Préciser le contenu de trois clauses:

   - Une clause PRE contenant un descriptif des conditions d'utilisation de la procédure.
   - Une clause MOD contenant les arguments modifiés.
   - Une clause ACT contenant un descriptif des résultats conséquents à l'action de la procédure. On fera aussi figurer un descriptif des transformations des arguments modifiés apparaissant dans la clause MOD.

   Les clauses pouvant éventuellement être vides.
   Nous donnons une présentation de ces clauses dans les exemples suivants:
   - (*booleen*) OCC (*tableau[entier]* A; *entier* e)
     PRE: A est trié par ordre croissant et ne contient pas de doublons.
     ACT: Retourne VRAI si e est dans A, FAUX sinon.
   - SUPP (*tableau[entier]* A; *entier* e).

PRE: A est trié par ordre croissant et ne contient pas de doublons.
MOD: A.
ACT: Supprime l'entier e si e est dans A.

- (*liste*) créer
  ACT: Crée une liste vide.

La mise en oeuvre d'une procédure doit répondre aux spécifications de son abstraction. Une fois vérifié le contenu de la clause PRE, elle doit en particulier ne modifier que les arguments apparaissant dans la clause MOD et agir en conformité avec le contenu de la clause ACT. Les procédures abstraites, une fois mise en oeuvre dans un langage, permettront une extension des opérations prédéfinies dans ce langage.
Remarques.
Dans certains situations il est nécessaire dans une procédure de prendre en charge des exeptions. Il est alors intéressant de faire apparaître ces exeptions dans les spécifications.Par exemple en reprenant la procédure SUPP et en supposant en plus que cette procédure retourne la chaine de caractère *vide* si le tableau A est vide, on aura:
SUPP (*tableau[entier]* A; *entier* e) (*chaine*)
PRE: A est trié par ordre croissant et ne contient pas de doublons.
MOD: A.
ACT: Supprime l'entier e si e est dans A,
sinon retourne la chaine de caractère *vide*.

6.1.2 Définition abstraite d'un nouveau type.

Un type abstrait de données (TAD) permet de caractériser la nature d'un groupe d'entités, ainsi que les opérations autorisées sur les entités de ce groupe. Par exemple le type *ensemble* d'éléments avec comme opérations autorisées sur un ensemble: la creation, l'insertion, la suppression, la taille, l'appartenance.
Un nouveau TAD doit répondre à des applications précises par exemple, un ensemble peut servir au stockage de données sans doublon et permettre en plus des opérations autorisées, d'autres opérations telles que la recherche d'un élément ou le tri.
Un TAD est adéquat lorsqu'il contient toute opération nécessaire à une utilisation dans une application donnée.
Un type de données demeure abstrait tant qu'il n'a pas été construit dans un langage.
L'abstraction de type permet de différer les décisions de choix de mise en oeuvre dans un langage, par exemple si l'on définit le TAD **mot**, on pourra choisir de l'implémenter soit par une chaîne de caractères, soit par un tableau de caractères.
Nous pouvons caractériser un type, de façon abstraite à partir de spécifications. Les règles que nous retiendrons pour spécifier un nouveau TAD sont les suivantes: - Préciser dans un en-tête: le nom du TAD, la liste des noms des procédures autorisées.
- Préciser le contenu d'une clause DESC, qui est un descriptif des entités ayant pour type le TAD.
- Préciser dans une clause OP, les spécifications de chacune des procédures autorisées.

6.1.3 Exemple:Le TAD *ensemble*.

ensemble (creer, inserer, oter, appartient, taille, choisir)

DESC: Tout entite de type ***ensemble*** est un ensemble non borne d'elements de meme type ***elt*** sans doublon. Le type ***elt*** etant deja defini.

OP:

(*ensemble*) creer $_p$

ACT: Retourne un ensemble vide

inserer $_p$(*ensemble* E; *elt* x)

MOD: E.

ACT: Insere x dans E.

oter $_p$(*ensemble* E; *elt* x)

MOD: E.

ACT: Supprime l'element x de E.

appartient $_p$(*ensemble* E; *elt* x)

ACT: Retourne VRAI si x est dans E, FAUX sinon

(*entier*) taille $_p$(*ensemble* E)

ACT: Retourne le nombre d'element de E.

(*elt*) choisir $_p$(*ensemble* E)

PRE: E est non vide.

ACT: Retourne un element quelconque de E.

Fig.5 Exemple 2: Le TAD *ensemble*.

Nous donnons en appendice à ce chapitre un manuel d'utilisation d'un langage algorithmique permettant d'intégrer dans des programmes (en langage algorithmique) des procédures ou des types définis abstraitement.

Exercice 1.

Définir abstraitement (spécifier) une procédure qui permettra de supprimer toute redondance d'éléments dans un ensemble et construire cette procédure, en utilisant le langage algorithmique.

ensemble rep(*ensemble* E)

MOD: E

ACT: Supprime les redondances d'éléments dans E.

Retourne E sans redondance d'éléments.

```
rep ( ensemble E )
  {
   elt x
   ensemble A
   A=creer p
   tant que taille p(E) ≠ 0
     {
       x=choisir p(E)
       oter p(E, x)
       si appartient p (A, x)=FAUX
          inserer p(A, x)
     }
   E=A
   retourner(E)
  }
```

Exercice 2.
Définir le type graphe à partir du type ensemble. On supposera défini abstraitement le type paire d'entiers.

```
type graphe =
     {
      ensemble[entier] SO
      ensemble[paire] AR
     }
```

6.2 Définition abstraite du type *graphe*.

graphe (creerg, aj-som, aj-arc, sup-som, sup-arc, ens-adj, videg)

DESC: Tout objet G de type {\it graphe} est caracterise par un ensemble de sommets X
et un ensemble d'arcs U dont les extremites sont dans X. On suppose definis.
les types *sommet* et *arc* .

OP:

graphe creerg$_p$()
ACT: Retourne un graphe vide

aj-sommet ($_p$ *graphe* G, *sommet* s)
MOD: G.
ACT: Insere un sommet dans l'ensemble X.

aj-arc ($_p$ *graphe* G, *sommet* s1, s2)
REQ: s1 et s2 sont dans X.
MOD: G.
ACT: Insere dans U, un arc dont les extremites sont s1 et s2

sup-som ($_p$ *graphe* G, *sommet* s)
MOD: G.
ACT: Supprime le sommet s de X ainsi que tous les arcs dans U dont l'une
des extremites est s.

sup-arc ($_p$ *graphe* G, *sommet* s1, s2)
MOD: G.
ACT: Supprime les arcs dans U ayant pour extremites s1 et s2.

ensemble ens-adj ($_p$ *graphe* G, *sommet* s)
REQ: s est dans X.
ACT: Retourne l'ensemble des sommets de G adjacents a s.

booleen videg$_p$(*graphe* G)
ACT: Retourne VRAI si G est vide, FAUX sinon.

Fig.5 Exemple 2: Le TAD *grahe*.

Pour les procédures que l'on va construire, voir la section précédente pour les principes des algorithmes.

6.3 L'algorithme de parcours en profondeur

On définit les trois propcédures suivantes :

Visiter(*Sommet* S; *Graphe* G)
MOD: S, G.
ACT: Ferme le sommet S, Effectue une opération sur ce sommet.
comme par exemple l'affichage des informations associées à S.

(*booléen*)ouvert(*sommet* S; *Graphe* G)
ACT: Retourne VRAI si S est ouvert, FAUX sinon.

(*sommet*) copie(*ensemble[elt]* E, *tableau[elt][n]* TAB)
REQ:taille$_p$(E)=n.
ACT: Recopie les élements de E dans TAB.

L'algorithme de parcours d'un graphe G à partir d'un de ses sommets V est alors le suivant :

Profondeur(*Graphe* G; *Sommet* V)
{
tableau[sommet][] TAB

```
        ensemble E=ens-adj_p(G, V)
        visiter(G, V)
        copie(E, TAB)
        Pour i=0 à taille_p(E)-1
          {
            si ouvert(V, G)
              {
                Profondeur(G,TAB[i])
              }
            i=i+1
          }
    }
```

Exercices.
1) Donner les traces d'exécution de ce programme avec en entrée le graphe de la figure 15 et le sommet a.
2) Ecrire une version itérative de Profondeur(). On pourra utiliser une pile pour stocker le sommet courant et sa liste d'ancêtres. Cela revient à simuler les différents appels récursifs. Pour implanter l'algorithme, on utilise une pile dans laquelle on place les sommets dont la liste d'adjacence reste à explorer.

6.4 L'algorithme de parcours en largeur

On définit les deux procédures suivantes :

(*Sommet*)oter(*tableau[sommet][]* FILE)
MOD: FILE
ACT: supprime le dernier élément de FILE et le retourne.

mettre(*tableau[sommet][]* FILE, *sommet* s)
MOD: FILE
ACT: Insère s à la première position dans FILE,
les autres positions dans FILE étant incrémentées d'une unité .

```
Largeur( Graphe G; Sommet V)
    {
        sommet s
        ensemble E
        tableau[sommet][] T, FILE
        mettre(FILE, V)
        visiter(G, V)
        tant que vide_p(FILE)=FAUX
            {
                s=oter(FILE)
                E=ens-adj_p(G, s)
                si taille_p(E)≠0
                    {
                    copie(E, TAB)
                    Pour i=0 à taille_p(E)-1
                        {
                            si ouvert(T[i], G)
                                {
                                    visiter(G, T[i])
                                    mettre(FILE,T[i])
                                }
                             i=i+1
                        }
                }
    }
```

Exercice.
Donner les traces d'exécution de ce programme avec en entrée le graphe de la figure 15 et le sommet a.

6.5 Recherche des composantes connexes.

On modifie la fonction Visiter() de l'algorithme Profondeur() de façon à ce qu'elle marque les sommets visités avec un numéro de composante connexe. La fonction de parcours s'écrira alors :

```
Profondeur2( Graphe G; Sommet V; entier NC)
    ⋮
    Si ...
        Visiter( G, V, NC )
        ⋮
            Profondeur2( G, S, NC )
```

L'algorithme de détermination des composantes connexes est alors le suivant :

```
Connexité( Graphe G ) ( Liste[ Sommet ] )
    Sommet S
    Liste[ Sommet ] L

    NC ← 0
    L ← ListSom( G )

    Pour chaque S dans L
        Si Ouvert( G, S ) = VRAI
            NC ← NC + 1
            Profondeur2( G, S, NC )
```

retourner L

L contient la liste des sommets, marqués par leur numéro de composante connexe.

Chapter 7

Quelques algorithmes dans les graphes

7.1 Problèmes de cheminement. Plus court chemin

On considère un graphe $G = (X, U)$. On associe à chaque arc $u \in U$ un nombre réel $l(u)$ appelé *longueur de l'arc.*

problème de plus court chemin Il existe de nombreuses applications aux problèmes de plus court chemin et à leurs variantes. Le problème de plus court chemin entre deux sommets i et j consiste à déterminer parmi tous les chemins joignant i et j, un chemin élémentaire μ dont la longueur totale $\sum_{u \in \mu} l(u)$ soit minimale ($u \in \mu$ signifie que l'arc u appartient au chemin μ).

On admettra le résultat suivant :

Une solution à ce problème existe ssi il n'existe pas dans le graphe de circuit de longueur strictement négative pouvant être atteint à partir de i.

Les algorithmes de cheminement qui sont présentés peuvent être divisés en deux grandes catégories :

- Ceux pour lesquels on recherche un plus court chemin d'un sommet spécifié i_0 à tous les autres sommets du graphe.
- Ceux dans lesquels on recherche un plus court chemin entre tout couple (i, j) de sommets du graphe.

On distingue parmi ces algorithmes ceux qui prennet en compte des longueurs quelconques pour les arcs et ceux pour lesquels les longueurs doivent être positives ou nulles. Plus les types de longueurs sont restreints, plus les algorithmes sont efficaces.

Un algorithme permettant de répondre au *problème de chemin de capacité maximale* est également présenté.

7.2 Algorithme de Ford-Bellman

L'algorithme de Ford-Bellman permet de trouver un ensemble de plus court chemins d'origine fixée i_0, dans un graphe pondéré de longueurs quelconques.

7.2.1 Principe de l'algorithme

L'algorithme de Ford-Bellman consiste essentiellement en une procédure de marquage dans laquelle on associe à chaque sommet i un nombre (*une marque*) $\pi(i)$. Ces marques sont modifiées à chaque étape de l'algorithme de telle sorte que à la fin de la procédure, $\pi(i)$ représente la longueur d'un plus court chemin entre i_0 et i. Les valeurs finales $\pi^*(i),\ i = [1..N]$ des longueurs d'un plus court chemin entre i_0 ($\pi(i_0) = 0$) et les $N - 1$ autres sommets d'un graphe $G = (X, U)$ sont caractérisés par les inégalités **(1)** :

$$\forall i,\ j \in U : \quad \pi^*(i) \leq \pi^*(j) + l_{ij}$$

où l_{ij} est la longueur de l'arc (i, j).

L'algorithme permet de parcourir la liste des arcs du graphe et à partir d'un ensemble provisoire de valeurs $\pi(i),\ i \in X$, de remplacer par $\pi(i) + l_{ij}$ celles des valeurs $\pi(j)$ pour lesquelles $\pi(j) > \pi(i) + l_{ij}$. Une telle modification signifie que l'on a amélioré $\pi(j)$ en empruntant un chemin de longueur $\pi(i)$ entre i_0 et i suivi de l'arc (i, j) de longueur l_{ij}. Le parcours du graphe G à partir de i_0 se fait autant de fois que nécessaire, jusqu'à obtention des longueurs $\pi(j), j \in X$ vérifiant toutes les inégalités **(1)**.

Les marques sont initialisées par $\pi(i_0) = 0$ et $\pi(i) = \infty,\ \forall i \in X, i \neq i_0$.

7.2.2 Complexité

La complexité de cet algorithme est facile à déterminer : en effet, chaque itération consiste en un parcours complet de la liste des arcs, ce qui nécessite $M = |U|$ opérations, et comme dans le pire des cas, il faut $N = |X|$ itérations, on en conclue que l'algorithme est de complexité $O(MN)$.

En admettant le résultat suivant

Si un graphe $G = (X, U),\ N = |X|$, n'admet pas de circuit de longueur strictement négative, alors les valeurs finales $\pi^*(i)$ sont obtenues en au plus $N - 1$ itérations.

on peut aussi utiliser l'algorithme de Ford-Bellman pour détecter un circuit de longueur strictement négative dans le graphe, suivant que le nombre d'itérations est inférieur ou supérieur à $N - 1$.

7.2.3 Enoncé formel de l'algorithme

Ford-Bellman

1– Initialisation

$\pi(i_0) \leftarrow 0, \pi(i) \leftarrow \infty,\ \forall i \in X, i \neq i_0$.
$\forall i \in X, p(i) \leftarrow i$, — le prédecesseur de i est i.
$k \leftarrow 0$ — compteur d'itérations.

2– Itération courante

Modif ← FAUX — aucune modification de $\pi(i)$ n'est possible.
$\forall i \in X,\ \ \forall j \in \Gamma_i$ — Γ_i est l'ensemble des successeurs de i
  $v \leftarrow \pi(i) + l_{ij}$
  Si $v < \pi(j)$
    $\pi(j) \leftarrow v$
    $p(j) \leftarrow i$
    Modif ← VRAI

3– Test de fin

Si Modif = FAUX — aucune amélioration des marques lors de l'étape k.
  fin
Sinon
  $k \leftarrow k + 1$
  Si $k = N$
    fin — car circuit de longueur négative.
  Sinon
    aller en 2– — itération suivante.

exemple On déroule l'algorithme de Ford-Bellman à partir du sommet x_0 du graphe de la figure 7.1, avec les initialisations $\pi(x_0) \leftarrow 0$, $\pi(x_i) \leftarrow \infty,\ \forall x_i \in X, x_i \neq x_0,\quad p(x_i) = x_i$.

On obtient $\pi(x_1) = 2, \pi(x_2) = 4, \pi(x_3) = 3, \pi(x_4) = 8$ pour les chemins $x_0x1,\quad x_0x_1x_3x_2,\quad x_0x_1x_3,\quad x_0x_1x_3x_2x_4$.

figure=r.ps,width=6cm,height=4cm

Etape	initial	terminal $\pi(j)$	$p(j)$
1	x_0	$\pi(x_1) = 2$	$p(x_1) = x_0$
		$\pi(x_2) = 5$	$p(x_2) = x_0$
	x_1	$\pi(x_3) = 3$	$p(x_3) = x_1$
		$\pi(x_4) = 9$	$p(x_4) = x_1$
	x_3	$\pi(x_2) = 4$	$p(x_2) = x_3$
2	x_2	$\pi(x_4) = 8$	$p(x_4) = x_2$
3	**fin.**		

Figure 7.1: Application de Ford-Bellman sur un graphe pondéré

7.3 Algorithme de Dijkstra

Cet algorithme permet de déterminer un ensemble de plus courts chemins à partir d'un sommet i_0 dans un graphe muni d'arcs de longueurs positives ou nulles.

7.3.1 Principe de l'algorithme

Comme dans l'algorithme de Ford-Bellman, on associe à chaque sommet i du graphe une marque $\pi(i)$ qui doit, en fin d'algorithme, représenter la longueur d'un plus court chemin de i_0 à i.

Cet algorithme tient à jour un ensemble E de sommets dont les distances les plus courtes à i_0 sont déjà connues. Au départ E contient uniquement i_0. A chaque étape, on ajoute à E l'un des sommets restant dont la distance à i_0 est la plus courte possible.

L'algorithme procède en $N-1$ itérations. Lors d'une itération quelconque, l'ensemble X des sommets du graphe est partagé entre E et son complémentaire. E contient l'ensemble des sommets i pour lesquells la marque $\pi(i)$ représente effectivement la longueur d'un plus court chemin entre i_0 et i. Comme les arcs ont une longueur positive ou nulle, il est possible de trouver un plus court chemin de i_0 à un sommet quelconque en passant par des sommets de E. On utilise un tableau D pour inscrire les longueurs de tels chemins. Dès que tous les sommets sont inclus dans E, D contient alors les distances des plus courts chemins entre la source et les autres sommets du graphe.

7.3.2 Enoncé formel de l'algorithme

Dijkstra

1– Initialisation

$E \leftarrow \{i_0\}$, $D[i] \leftarrow l_{i_0 i}$ si l'arc (i_0, i) n'existe pas, $l_{i_0 i} \leftarrow \infty$.

2– Itération courante

choisir $j \notin E$ tel que $D[j]$ soit minimum.

$E = E \bigcup \{j\}$

Pour chaque k tel que $k \notin E$

$D[k] \leftarrow min(D[k], D[j] + l_{jk})$

exemple On déroule l'algorithme de Dijkstra sur le graphe de la figure 7.2. Le sommet source est x_0.

figure=s.ps,width=4cm,height=4cm

Etape	E	Sommet choisi	$D[2]$	$D[3]$	$D[4]$	$D[5]$
0	$\{x_1\}$	-	10	∞	30	100
1	$\{x_1, x_2\}$	x_2	10	60	30	100
2	$\{x_1, x_2, x_4\}$	x_4	10	50	30	90
3	$\{x_1, x_2, x_4, x_3\}$	x_3	10	50	30	60
4	$\{x_1, x_2, x_4, x_3, x_5\}$	x_5	10	50	30	60

Figure 7.2: Application de Dijkstra sur un graphe pondéré

Si l'on désire retrouver un plus court chemin de la source à chaque sommet, on maintient en plus un tableau C de sommets tel que $C[j]$ contienne le prédecesseur de i dans un plus court chemin de i_0 à j. C est initialisé par $C[j] = i_0, \forall j \neq i_0$. Pour la mise à jour des chemins, il faut ajouter à l'étape courante :
Si $D[j] + l_{jk} < D[k]$
$C[k] \leftarrow j$

exemple Pour l'exemple de la figure 7.2, on obtient

$$C[2] = 1, \quad C[3] = 4, \quad C[4] = 1, \quad C[5] = 3$$

d'où un plus court chemin de x_1 à x_5 est $x_1 x_4 x_3 x_5$.

7.4 Chemin de capacité maximale

7.4.1 Définition

• *Capacité*
On considère un graphe orienté $G = (X, U)$. A chaque arc $u \in U$, on associe un nombre réel c_u appelé *capacité* de l'arc u (en général, $c_u \geq 0$). Etant donné un chemin élémentaire $C = u_1 u_2 ... u_p$ entre deux sommets i et j de G, on appelle capacité du chemin C la quantité $c(C) = min_{u \in C}\ c_u$.
• *Problème du chemin de capacité maximale*
Le problème du chemin de capacité maximale consiste à rechercher parmi l'ensemble des chemins possibles un chemin de capacité maximale.

Nous nous intéressons à la recherche des chemins de capacité maximale entre un sommet i_0 donné et les autres sommets du graphe G.

7.4.2 Remarques

• On peut toujours considérer que les chemins optimaux cherchés sont élémentaires (ne contiennent pas de circuits). En effet si tel n'est pas le cas, supposons que un chemin optimal P_j^* entre i_0 et j ne soit pas élémentaire, alors il contient au moins un circuit. En éliminant certains arcs, on peut éliminer le circuit et construire un nouveau chemin de i_0 à j de capacité au moins égale à celle de P_j^*.

• Soit un chemin élémentaire optimal P_j^* entre i_0 et j. Si i est sur le chemin P_j^*, alors le chemin de i_0 à i (noté P_i^*) contenu dans P_j^* est aussi un chemin optimal.

• Si on note Γ_j^{-1} l'ensemble des prédécesseurs d'un sommet j du graphe, alors on a la relation **i** :

$$\forall j \neq i_0, \quad c_j^* = Max_{i \in \Gamma_j^{-1}}\ Min\{c_i^*, c_{ij}\} \qquad et \qquad c_{i0} = \infty$$

$c_j^* = c(P_j^*)$ désigne les valeurs optimales des capacités des chemins de i_0 à j lorsque j parcourt l'ensemble des sommets distincts de i_0.

On peut remarquer l'analogie avec les équations du problème de plus court chemin **ii** :

$$\forall j \neq i_0, \quad \pi^*(j) = Min_{i \in \Gamma_j^{-1}}\ \{\pi^*(i) + l_{ij}\} \qquad et \qquad \pi^*(i_0) = 0$$

• Les équations **i** peuvent être considérées comme constituant un système d'équations linéaires

$$\forall j \neq i_0, \quad c_j^* = \bigoplus_{i \in \Gamma_j^{-1}} \bigotimes (c_i^*, c_{ij})$$

en prenant comme opérations algébriques, les opérations définies par :

$$\forall a, b \in \mathcal{R} \quad \begin{array}{l} a \bigoplus b = Max\{a, b\} \\ a \bigotimes b = Min\{a, b\} \end{array}$$

et $-\infty$, $+\infty$ éléments neutres de $\bigoplus$ et $\bigotimes$ respectivement.

L'analogie des systèmes **i** et **ii** permet de construire un algorithme adapté au problème de capacité maximale à partir de l'algorithme de Dijkstra.

7.4.3 Enoncé formel de l'algorithme

Capacité
1– Initialisation
$E \leftarrow \{i_0\}, \quad D[i] \leftarrow c_{i_0 i}$ $c_{i_0 i}$ est la capacité de l'arc (i_0, i) si il existe, 0 sinon.

2– Itération courante
choisir $i \notin E$ tel que $D[i]$ soit maximum.
$E = E \bigcup \{i\}$
Pour chaque j tel que $j \notin E$
$D[j] \leftarrow max(D[j],\ min(D[i], c_{ij}))$

Si $E = X$
fin

7.5 Plus court chemin entre toutes les paires de sommets : Floyd

7.5.1 Principe de l'algorithme

On remarque tout d'abord que si tous les arcs sont munis de longueurs positives ou nulles, il suffit d'appliquer N fois l'algorithme de Dijkstra. Si les certains arcs sont de longueur strictement négative, on peut utiliser N fois l'algorithme de Ford-Bellman.

Cependant, on préfère utiliser un algorithme associé à une méthode *matricielle*. Notons L la $N \times N$ matrice $(l_{ij})_{1 \leq i\ j \leq N}$ dont le terme l_{ij}, $\ i \neq j$ est égal à la longueur de l'arc (i, j) si il existe, $+\infty$ sinon. Pour les éléments diagonaux $(i = j)$, $l_{ij} = 0$.

Pour $1 \leq k \leq N$, notons L^k la matrice $(l^k_{ij})_{ij}$ dont le terme (l^k_{ij}) désigne la longueur minimale d'un chemin d'origine i et d'extrémité j dont tous les sommets intermédiaires appartiennent au sous-ensemble $\{1, 2, ..., k\}$. Lorsque $i = j$, (l^k_{ij}) désigne la longueur minimale d'un circuit passant par i et astreint à n'utiliser comme sommets intermédiaires que des sommets parmi les sommets 1, 2, ..., k.

Pour $k = 0$, on a $L^0 = L$.

On remarque que alors les coefficients des matrices L^k sont liés par la relation de récurrence **R** :

$$\forall i, \forall j, \quad l^k_{ij} = Min\{l^{k-1}_{ij},\ l^{k-1}_{ik} + l^{k-1}_{kj}\}$$

En effet, deux situations peuvent se présenter :

- Le plus court chemin de i à j avec les sommets $\{1, 2, ..., k\}$ passe par k.

  Dans ce cas, le chemin considéré est formé d'un sous-chemin entre i et k, suivi d'un sous-chemin entre k et j. Ces sous-chemins ne doivent contenir que des sommets dans $\{1, 2, ..., k-1\}$ et être de longueur minimale. On aura donc :
  $$l^k_{ij} = l^{k-1}_{ik} + l^{k-1}_{kj}$$

- Le plus court chemin de i à j avec les sommets $\{1, 2, ..., k\}$ ne passe par k.

  Dans ce cas, on aura évidemment
  $$l^k_{ij} = l^{k-1}_{ij}$$

La matrice L^N, donnant l'ensemble des plus courts chemins dans le graphe est déterminée en N étapes par récurrence à partir de L grâce à la relation **R**. Cet algorithme permet de détecter la présence d'un circuit strictement négatif dès que l'un des l_{ii}^k est strictement négatif.

Cet algorithme peut être complété par la construction de la matrice prédécesseur $A = (a_{ij})_{1 \leq i,\ j \leq N}$ dont les termes a_{ij} ont la signification suivante : *a_{ij} est le numéro du sommet prédécesseur de j sur le plus court chemin entre i et j trouvé jusqu'à présent*. Cette matrice est mise à jour par :

$a_{ij} = a_{kj}$ si $l_{ij}^k = l_{ik}^{k-1} + l_{kj}^{k-1}$

a_{ij} reste inchangé si $l_{ij}^k = l_{ij}^{k-1}$

à chaque application de la relation de récurrence **R**.

7.5.2 Enoncé formel de l'algorithme

Floyd

1– Initialisation

Pour $i \leftarrow 1, 2, ..., N$ et $j \leftarrow 1, 2, ..., N$

$l_{ij} \leftarrow longueur(i, j)$ si (i, j) existe.

$l_{ij} \leftarrow +\infty$ sinon.

$l_{ii} \leftarrow 0$

$a_{ij} \leftarrow i$

2– Itération courante k, $k = [1..N]$

Pour $i \leftarrow 1, 2, ..., N$ et $j \leftarrow 1, 2, ..., N$

$v \leftarrow l_{ik} + l_{kj}$

Si $v < l_{ij}$

$l_{ij} \leftarrow v$

$a_{ij} \leftarrow a_{kj}$

Si $i = j$ et $l_{ij} < 0$ il existe un circuit de longueur strictement négative

fin

exemple On applique l'algorithme de Floyd au graphe de la figure 7.3.

figure=t.ps,width=6cm,height=4cm

Figure 7.3: Un graphe pour l'algorithme de Floyd

L'algorithme de Floyd se déroule suivant les étapes :

Initialisation : $\begin{bmatrix} 0 & 3 & +\infty & 3 \\ 2 & 0 & 2 & 2 \\ -2 & +\infty & 0 & 1 \\ +\infty & 4 & 4 & 0 \end{bmatrix}$
Etape 1 : $\begin{bmatrix} 0 & 3 & +\infty & 3 \\ 2 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ +\infty & 4 & 4 & 0 \end{bmatrix}$
Etape 2 : $\begin{bmatrix} 0 & 3 & 5 & 3 \\ 2 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ 6 & 4 & 4 & 0 \end{bmatrix}$
Etape 3 : $\begin{bmatrix} 0 & 3 & 5 & 3 \\ 0 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ 2 & 4 & 4 & 0 \end{bmatrix}$
Etape 4 : $\begin{bmatrix} 0 & 3 & 5 & 3 \\ 0 & 0 & 2 & 2 \\ -2 & 1 & 0 & 1 \\ 2 & 4 & 4 & 0 \end{bmatrix}$

Les matrices A correspondantes sont les suivantes :

Initialisation : $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$
Etape 1 : $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$
Etape 2 : $\begin{bmatrix} 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 \\ 2 & 4 & 4 & 4 \end{bmatrix}$
Etape 3 : $\begin{bmatrix} 1 & 1 & 2 & 1 \\ 3 & 3 & 3 & 3 \\ 3 & 1 & 3 & 3 \\ 3 & 4 & 4 & 4 \end{bmatrix}$
Etape 4 : $\begin{bmatrix} 1 & 1 & 2 & 1 \\ 3 & 3 & 3 & 3 \\ 3 & 1 & 3 & 3 \\ 3 & 4 & 4 & 4 \end{bmatrix}$

exemple Si l'on veut par exemple retrouver le chemin de longueur minimale entre 4 et 1, on a comme premier prédécesseur de 1 dans ce chemin le sommet 3 ($A_{41} = 3$) et comme premier prédécesseur de 3 le sommet 4 ($A_{43} = 4$). On a donc le chemin 4 3 1.

7.6 Arbre de poids minimal – Test de connexité

7.6.1 Définition

Etant donné un graphe non orienté $G = (X, U)$, on dit que le graphe partiel $H = (X, T)$, $U \supseteq T$ est une *forêt* de G si il ne contient pas de cycle. H est une *forêt maximale* si il est sans cycle et que aucune arête de U ne peut être ajoutée à T sans créer de cycle. On démontre que

Si G a N sommets et M arêtes, alors toute forêt maximale de G contient nécessairement $N - p$ arêtes, où p est le nombre de composantes connexes de G.

Dans le cas où G est connexe ($p = 1$), un *arbre* de G est défini comme étant une forêt maximale. De plus, tout graphe connexe contient au moins un arbre.

7.6.2 Principe de l'algorithme de Prim

La structure de l'algorithme est la même que pour l'algorithme de Dijkstra. A chaque étape, on ajoute un sommet à l'arbre de poids minimum en construction. Le tableau des marques π est tel que $\pi(i)$ est le poids minimal d'une arête $u = (i, j)$ connectant i à un sommet j de l'arbre. Le tableau a contient cette arête u de connexion de i à l'arbre ($a(i) = \alpha u$). Deux cas peuvent se présenter :

1 i n'est pas encore inclus. Dans ce cas $\alpha = -1$

2 i est déjà inclus dans l'arbre. Dans ce cas $\alpha = +1$

A chaque étape, le sommet i choisi est celui non inclus ($a(i) < 0$) le plus proche de l'arbre ($\pi(i)$ minimum). Lors de l'inclusion de i, les informations concernant les sommets non inclus adjacents à i sont mises à jour.

A la fin de l'algorithme, a contient les arêtes incluses dans l'arbre de poids minimum et π les poids correspondant.

remarques Un sommet arbitraire est utilisé comme point de départ de l'arbre, sans arête associée. Comme un arbre de poids minimum sur le graphe $G = (X, U)$ comporte au plus $N - 1$ arêtes ($N = |X|$) (voir ci-dessus), a et π sont de taille N.

7.6.3 Enoncé formel de l'algorithme de Prim

Prim

1– Initialisation

$\forall i \in X \quad \pi(i_0) \leftarrow -\infty, \quad \pi(i) \leftarrow +\infty$ si $i \neq i_0$

$\forall i \in X \quad a(i) \leftarrow -\infty$

$A \leftarrow \emptyset$

2– Itération courante

Sélectionner $i \in X - A$ tel que $\pi(i) = Min_{j:a(j)<0}\pi(j)$ — minimum pour j dans $X - A$

Si $\pi(i) = +\infty$ ou $X = A$

  fin

Sinon

  $a(i) \leftarrow -a(i)$

Pour toute arête $u = (i, j)$ ayant i comme extrémité — mise à jour des marques

  Si $w_u < \pi(j)$ et $a(j) < 0$

    $\pi(j) \leftarrow w_u$

    $a(j) \leftarrow -u$

exemple

Graphe	It.	A / $\bar{A}$	π / a					
figure=u0.ps,width=4cm,height=3cm	init	$\emptyset$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
		X	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
figure=u1.ps,width=4cm,height=3cm	1	$\{1\}$	$-\infty$	6	1	5	$+\infty$	$+\infty$
		$\{2,3,4,5,6\}$	$-\infty$	-1	-2	-3	$-\infty$	$-\infty$
figure=u2.ps,width=4cm,height=3cm	2	$\{1,3\}$	$-\infty$	5	1	5	6	4
		$\{2,4,5,6\}$	$-\infty$	-4	2	-3	-7	-8
figure=u3.ps,width=4cm,height=3cm	3	$\{1,3,6\}$	$-\infty$	5	1	2	6	4
		$\{2,4,5\}$	$-\infty$	-4	2	-9	-7	8
figure=u4.ps,width=4cm,height=3cm	4	$\{1,3,4,6\}$	$-\infty$	5	1	2	6	4
		$\{2,5\}$	$-\infty$	-4	2	9	-7	8
figure=u5.ps,width=4cm,height=3cm	5	$\{1,3,4,5,6\}$	$-\infty$	5	1	2	3	4
		$\{2\}$	$-\infty$	4	2	9	-6	8
figure=u6.ps,width=4cm,height=3cm	6	X	$-\infty$	5	1	2	3	4
		$\emptyset$	$-\infty$	4	2	9	6	8

7.7 Problème de flot

7.7.1 Définition

Soit $G = (X, U)$ un graphe orienté. On associe à chaque arc $u \in U$ un nombre réel, noté f_u tel que en tout sommet $i \in X$ soit vérifiée l'égalité **I** :

$$\sum_{u \in \omega^+(i)} f_u \quad = \quad \sum_{u \in \omega^-(i)} f_u$$

Pour tout sommet u dans U, f_u est appelée *quantité de flot* ou *flux* sur l'arc u.

L'égalité **I** exprime que, en un sommet i du graphe, la somme des flux sortant $\sum_{u \in \omega^+(i)} f_u$ est égale à la somme des flux entrant $\sum_{u \in \omega^-(i)} f_u$.

On remarquera l'analogie entre l'égalité **I** et la loi de conservation aux noeuds de Kirchoff dans un réseau électrique. Pour cette raison, l'égalité **I** est appelée loi de Kirchoff dans les graphes.

exemple Dans le graphe de la figure 7.4, le vecteur $f = [5, 3, 2, -2, -1, 4]$ est un flot sur G. L'orientation choisie pour les arcs est arbitraire. La négativité d'une composante, f_4 par exemple, indique que le flot va du sommet 2 au sommet 3 sur l'arc 4.

figure=v.ps,width=6cm,height=4cm

Figure 7.4: Un flot sur un graphe. Les flux sont donnés entre parenthèses à côté des numéros des arcs.

7.7.2 Problème du flot maximum

On considère un graphe $G = (X, U)$ où chaque arc $u \in U$ est muni d'un nombre $c_u \geq 0$ appelé *capacité* de l'arc u. Lorsqu'on définit un flot sur G, c_u indique la limite supérieure d'un flot *admissible* sur u.

On se donne deux sommets distincts e et s dans G (e n'a pas de prédécesseur et constitue une entrée du graphe, s n'a pas de successeur et constitue une sortie du graphe).

exemple Dans l'exemple de la figure 7.4, $e = 1$ et $s = 4$. On considère le graphe $G^0 = (X, U^0)$ obtenu en ajoutant à G un arc fictif de e vers s. Dans l'exemple, il s'agit de l'arc 1.

- (e, s) est appelé *arc de retour de flot* de G^0. Il sera noté 0 par la suite.
- e est appelé sommet *source*.
- s est appelé sommet *puits*.

$f = [f_1, f_2, ..., f_M]$ est un *flot de e à s dans G* ssi l'égalité **I** est vérifiée. f_0 est appelée *valeur du flot*.

On remarque que si $f = [f_1, f_2, ..., f_M]$ est un flot de e à s dans G, $f' = [f_0, f_1, f_2, ..., f_M]$ est un flot dans G^0.

- *Problème de flot maximum*

Le problème de flot maximum de e à s dans G muni des capacités $c_u, \ \ u \in U$ est le suivant :
Déterminer un flot $f' = [f_0, f_1, f_2, ..., f_M]$ dans G^0 vérifiant les capacités de contraintes $0 \leq f_u \leq c_u, \ \ u = [1..M]$ *tel que la composante f_0 sur l'arc de retour (i.e. la valeur du flot) soit maximale*

7.7.3 Graphe d'écart et algorithme de Ford-Fulkerson

Soit $G = (X, U)$ un graphe muni de capacités c_u. Soit f un flot dans G. Au couple (G, f) on associe un graphe d'écart $G^{\Delta}(f) = (X, U^{\Delta}(f))$.

$U^{\Delta}(f)$ est défini comme suit. A tout arc $u = (i, j)$ de G, de capacité c_u, de flux f_u, on associe dans $G^{\Delta}(f)$:

- un arc $u^+ = (i, j)$ de capacité $c^{\Delta}(i, j) = c_u - f_u$ si l'arc u n'est pas *saturé*, c'est à dire si $f_u < c_u$. $c^{\Delta}(i, j)$ représente la capacité *résiduelle* de l'arc u dans G.
- un arc $u^- = (j, i)$ de capacité $c^{\Delta}(j, i) = f_u$ si le flux sur l'arc u n'est pas nul.

exemple La figure 7.5(a) représente un couple (G, f). Pour chaque arc, le couple *(capacité, flux)* est indiqué. La figure 7.5(b) représente le graphe d'écart $G^{\Delta}(f)$ associé.

figure=w0.ps,width=6cm,height=4cm figure=w1.ps,width=6cm,height=4cm

Figure 7.5: (a) Flot sur un graphe (b) Graphe d'écart associé

7.7.4 Algorithme de Ford-Fulkerson

L'intérêt du graphe d'écart apparait dans l'utilisation du résultat suivant :

> *Soit f un flot de e à s dans un graphe G, compatible avec les capacités c_u, i.e* $0 \leq f_u \leq c_u$. *Soit $G^{\Delta}(f)$ le graphe d'écart associé à G. Une condition nécessaire et suffisante pour que le flot f soit maximal sur G est qu'il n'existe pas de chemin de e à s dans $G^{\Delta}(f)$.*

Ce résultat est à la base de l'algorithme de Ford-Fulkerson.

7.7.5 Enoncé formel de l'algorithme

Ford-Fulkerson
1– Initialisation
$f \leftarrow (0, 0, ..., 0)$ flot courant

2– Itération courante
Trouver un chemin X de e à s dans $G^{\Delta}(f)$.
Si X non trouvé f est maximum
fin
Sinon
$r \leftarrow Min_{u \in X} c^{\Delta} u$ r est le minimum des capacités résiduelles des arcs du chemin.
Mise à jour de f :
$f_0 \leftarrow f_0 + r$
Si $u^+ \in X$
$f_u \leftarrow f_u + r$
Si $u^- \in X$
$f_u \leftarrow f_u - r$

7.8 K-coloration de graphe

7.8.1 Définition

On appelle *K-coloration* d'un graphe $G = (V, E)$, la répartition des sommets de V en k ensembles disjoints $X_1, X_2, \ldots, X_k$ tels que :

- $\forall v \in V, \quad v \in X_i \Rightarrow C_v = i, \quad i \in [\ 1, \ldots, k\ ]$. i correspond à la couleur affectée au sommet v (il y a une couleur par ensemble).
- $\forall e = (v, w) \in E, C_v \neq C_w$. Deux sommets adjacents doivent être coloriés par des couleurs différentes.

Le nombre de couleurs minimal nécessaire au coloriage d'un graphe correspond au *nombre chromatique* $\gamma(G)$ de ce graphe. Un graphe qui admet une coloration en k couleurs est dit *k-coloriable*. Le nombre chromatique est de 2 pour les graphes bipartis. Il est par exemple de 3 pour le graphe de Petersen (figure 7.6.

figure=color.ps,width=8cm,height=4.4cm

Figure 7.6: Exemples de coloriage : (a) graphe biparti. (b) graphe de Petersen.

Un algorithme simple pour trouver un coloriage valide (qui donne donc une borne supérieure sur le nombre chromatique du graphe est le suivant). La coloration est définie par les couleurs (entre 1 et *couleurs*) attribuées aux différents sommets et stockée, pour chaque sommet v dans C_v :

Color(G)

1– Initialisation

  couleurs $\leftarrow 0$ nombre de couleurs utilisées

  Pour tout sommet v de G, $C_v = 0$ sommets non coloriés

2– Itération courante, tant que tous les sommets ne sont pas coloriés.

  Trouver un sommet v non colorié dans G

  li_adj = $\leftarrow$ ListSadj(v, g)

  Si il existe une couleur $1 \leq c \leq couleurs$ t.q. $\forall s \in li_adj, \quad C_s \neq c$

    $C_V = c$

  sinon

    $couleurs \leftarrow couleurs + 1$

    $C_v = couleurs$

  retourner *couleurs*

Cet algorithme a l'inconvénient de ne pas déterminer un coloriage optimal, correspondant au nombre chromatique du graphe.

Nous explicitons ci-dessous un algorithme par énumération qui permet de déterminer tous les coloriages d'un graphes en au plus M couleurs.

7.8.2 Principe de l'algorithme par énumération

Déterminer pour un graphe quelconque un coloriage de taille minimale est un problème NP-complet. On peut utiliser pour trouver l'ensemble des coloriages en M couleurs au plus d'un graphe un algorithme par *énumération implicite*, qui examine l'ensemble des solutions (coloriages possibles), ou au moins, pour certaines d'entre elles, montre que il n'est pas possible d'obtenir un coloriage valide si certains sommets sont déjà colorés d'une certaine manière (d'où le terme *implicite*).

exemple Supposons que $M = 2$ (2 couleurs autorisées au maximum). A une étape quelconque de l'algorithme, on a le coloriage défini sur la figure 7.7 (a). Les sommets en blanc n'ont pas alors de couleur définie. On peut voir que pour le sommet c, il n'est pas possible de choisir une couleur (raies verticales ou horizontale), sans violer la contrainte de k-coloration, soit par rapport à l'arête $a \leftrightarrow c$, soit par rapport à l'arête $b \leftrightarrow c$.

figure=excolor.ps,width=3.5cm,height=2.5cm

Figure 7.7: Exemple de début de coloriage par énumération ($M = 2$)

Il n'est donc pas nécessaire d'aller attribuer des couleurs aux autres sommets non colorés. Il faut modifier la couleur de a ou la couleur de b.

L'algorithme génére l'ensemble des solutions possibles en M couleurs au plus de la manière suivante : l'arbre des solutions est construit en attribuant successivement les M couleurs à chaque sommet. Par exemple, pour $M = 2$ et $|V| = 5$ (nombre de sommets), l'arbre de recherche complet, avec les couleurs blanc et noir, est décrit sur la figure 7.8.

figure=grrech.ps,width=10cm,height=5cm

Figure 7.8: Arbre de recherche complet pour un graphe à 5 sommets, en 2 couleurs ($|V| = 2, \ M = 2$)

La figure 7.9 montre un exemple de recherche, pour $M = 2$ et $|V| = 5$. Le graphe à colorer est représenté en (a), et l'arbre de recherche effectif pour ce graphe est montré en (b). Une branche *coupée* dénote l'impossibilité de colorer le sommet suivant avec une couleur donnée.

Les nombre d'arêtes effectivement explorées dépend de l'ordre dans lequel les sommets sont ordonnés (v_1, v_2, v_3, v_4, v_5 dans l'exemple précédent). Il est possible que le nombre d'arêtes de l'arbre de recherche soit réduit en ordonnant l'arbre de recherche différement.

figure=grrechcut.ps,width=10cm,height=5cm

Figure 7.9: Arbre de recherche (b) d'un graphe à 5 sommets (a), en 2 couleurs ($|V| = 2,\ M = 2$)

exercice Déterminez l'arbre de recherche que l'on obtient en ordonnant les sommets pour l'exemple du graphe de la figure 7.9 (a) en v_1, v_4, v_5, v_2, v_3. Que pensez-vous du résultat ?

7.8.3 Ennoncé formel de l'algorithme

Pour définir l'ensemble des coloriages possibles d'un graphe $G = (V, E)$ en au plus M couleurs, l'algorithme utilise un vecteur $couleurs(1, ..., |V|)$ qui stocke l'indice correspondant à la couleur attribuée à un sommet (comprise entre 1 et M). Si un sommet i n'a pas encore de couleur attribuée, $couleurs(i) = 0$. Une procédure suivant($couleurs, i$) détermine et attribue la prochaine couleur valide pour le sommet i Si aucune couleur valide ne peut être attribuée au sommet (suivant les couleurs déjà attribuées aux sommets adjacents à i), $couleurs(i) = M + 1$.

Enum_Color($G = (V, E), M$**)**

1– Initialisation

- $couleurs(1, ..., |V|) \leftarrow 0$ — aucune couleur attribuée
- $k \leftarrow 0$
- $fin \leftarrow FAUX$

2– Itération courante.

Faire

suivant($couleurs, k$)

cas 1 : $couleurs(k) \leq M$ et $k < |V|$

$k \leftarrow k + 1$ — avancer dans l'arbre

cas 2 : $couleurs(k) \leq M$ et $k = |V|$

montrer($couleurs$) — une solution trouvée

cas 3 : $couleurs(k) > M$ et $k > M$

$couleurs(k) \leftarrow 0$ — impossible de colorier

$k \leftarrow k - 1$ — retour dans l'arbre

cas 4 : $couleurs(k) > M$ et $k = 1$

$fin \leftarrow VRAI$ — retour à la racine

jusqu'à $fin = VRAI$

retourner $couleurs$

exemple L'exécution de l'algorithme est illustrée pour le graphe de la figure 7.9 (a) dans le tableau ci-dessous avec $M = 3$. Les 18 premières itérations de l'algorithme sont représentées. Les colonnes v_i représentent $couleurs(i)$ à la fin de chaque étape. La colonne k montre la valeur de k à la fin d'une étape. la colonne cas montre l'action effectuée dans cette étape. Si l'on est dans le cas 3, suivant retourne $M + 1$ (= 4), et la couleur du sommet est remise à 0 (indiqué par 4(0) pour la couleur du sommet.

étape	v_1	v_2	v_3	v_4	v_5	cas	k
0	0	0	0	0	0	1	0
1	1	0	0	0	0	1	2
2	1	1	0	0	0	1	3
3	1	1	2	0	0	1	4
4	1	1	2	2	0	1	5
5	1	1	2	2	3	2	5
6	1	1	2	2	4(0)	3	4
7	1	1	2	3	0	1	5
8	1	1	2	3	4(0)	3	4
9	1	1	2	4(0)	0	3	3
10	1	1	3	0	0	1	4
11	1	1	3	2	0	1	5
12	1	1	3	2	4(0)	3	4
13	1	1	3	3	0	1	5
14	1	1	3	3	2	2	5
15	1	1	3	3	4(0)	3	4
16	1	1	3	4(0)	0	3	3
17	1	0	4(0)	0	0	3	2
18	1	2	0	0	0	1	3
				....			

Exercice Dessiner les solutions correspondant aux solutions trouvées et l'arbre de recherche défini par les étapes présentées dans le tableau. Décrivez les étapes suivantes de l'exécution de l'algorithme.

7.8.4 Une application : Emploi du temps

La coloration de graphes peut être utilisée pour modeliser des problèmes d'emploi du temps avec contraintes sur les ressources. Supposons un ensemble de tâches $T_1, ..., T_n$ à réaliser. Chaque tâche nécessite une unité de temps pour son exécution. Certaines tâches ne peuvent être réalisées simultannément car elles nécessitent la même ressource. Soit $C(1..n, 1..n)$ la matrice de conflit entre tâches : si $C(i,j) = 1$, les tâches nécessitent une ressource commune ($C(i,j) = 0$ dans le cas contraire). La matrice de conflit peut être modélisée comme un graphe $G = (V, E)$ comme suit :

- V est l'ensemble des tâches,
- $e = (i,j) \in E$ si $C(i,j) = 1$.

Le problème de trouver un *ordonnancement* des tâches correspond à déterminer une date d'exécution pour chacune d'entre elles, tel que 2 tâches en conflit n'est pas la même date d'exécution et que le temps total d'exécution soit minimisé.

Ce problème peut être résolu en déterminant une coloration des sommets du graphe de modélisation. Chaque couleur correspond à une date d'exécution distincte. La minimisation du nombre de couleurs permet de minimiser le nombre de dates, i.e. le temps global de l'ordonnancement. Une telle modélisation est illustrée sur la figure 7.10 pour une liste de 4 tâches présentant les conflits :

- $T_1 : T_2, T_3$
- $T_2 : T_1, T_3$
- $T_3 : T_2, T_3, T_4$
- $T_4 : T_3$

figure=colorsched.ps,width=3cm,height=2.5cm

Figure 7.10: Modélisation d'un problème d'ordonnancement avec contraites de ressources pour 4 tâches

Une solution est $(T_1, T_4), (T_2), (T_3)$.

exercice Retrouvez une solution à l'aide de l'algorithme par énumération donné ci-dessus ($M = 3$).

exercice Les étudiants a, b, c et d doivent subir un examen d'informatique, les étudiants a, e, f et g une épreuve de biologie, les étudiants b, c, g une épreuve de lettres, a, d, f une épreuve de sciences sociales, b et c doivent aller à un examen d'histoire.

Chaque épreuve a lieu une seule fois et dure la journée. Modéliser le problème en tant que problème de k-coloration de graphes. Trouvez le nombre de jours minimal pour faire passer les examens.

Chapter 8

Ordonnancement

8.1 Introduction

" Ordonnancer c'est programmer la réalisation d'un projet en attribuant des ressources aux tâches et en fixant leur date d'exécution."

De nombreux projets liés à des activités sociales ou économiques, demandent une organisation et une coordination parfaite pour leur réalisation. Les problèmes qui apparaissent en cours de réalisation, rejoignent dans la quasi-totalité des cas des problèmes d'ordonnancement.

exemple Chantier de construction, exécution de tâches sur des machines dans un atelier, ordonnancement de tâches sur plusieurs processeurs.

En toute généralité, les problèmes d'ordonnancement se posent en ces termes:

Etant donné un objectif qu'on se propose d'atteindre et dont la réalisation suppose:

- l'existence de ressources,

- l'exécution de tâches soumises à de nombreuses contraintes,

déterminer, l'ordre et le calendrier d'exécution des différentes tâches afin d'optimiser une fonction économique.

- Un problème d'ordonnancement peut être *statique*, lorsque l'ensemble des tâches à accomplir au cours du temps est connu au départ ainsi que les dates de début d'exécution des tâches.
- Un problème sera *dynamique* si l'ensemble des tâches à accomplir au cours du temps évolue de façon non déterministe.
- Quand les paramètres d'une tâche sont connus en probabilité, on dit que le problème considéré est *stochastique.*

Examinons de façon plus détaillé, les différentes notions introduites précédemment.

8.1.1 Les contraintes

On considère trois types de contraintes:

1. Les contraintes de type *potentiel* qui peuvent être des contraintes :
   - de succession: Une tâche B ne peut commencer avant qu'une tâche A ne soit terminée ou parvenue à un degré d'achèvement.
   - de localisation temporaire: Une tâche A ne doit pas commencer avant telle date ou encore doit être achevée à telle date.

2. Les contraintes de type *disjonctif*, lorsqu'on impose la réalisation non simultanée de certaines paires de tâches.

3. Les contraintes de type *cumulatif*, lorsqu'au cours du temps, il y a accumulation de moyens consacrés à la réalisation de tâches.

Dans notre cours nous nous préoccuperons essentielllement de contraintes de type potentiel ou disjonctif.

8.1.2 Les tâches

Dans un problème d'ordonnancement simple, une tâche est exécutée une seule fois et entièrement. Cependant dans la pratique plusieurs situations peuvent être rencontrées :

1. Une tâche peut être exécutée par morceau, on dit alors que la tâche est *morcelable* ou que la *préemption* est possible. Une tâche A est prémptée par une tâche B à l'instant t si A était en cours d'exécution avant t et employait une ressource allouée à B à l'instant t, ce qui provoque l'interruption de A.

2. Plusieurs tâches $A, B, ...$ peuvent être soumises à des contraintes de ressource et à des *contraintes internes* de succession.
   A donnant naissance à une infinité de tâches $A_1, A_2, ...A_h, ...$
   B donnant naissance à une infinité de tâches $B_1, B_2, ...B_k, ...$
   Une contrainte interne s'exprime par le fait que la tâche A_h précéde la tâche B_h.

   Une *contrainte externe* s'exprime par le fait que: $\exists k, \quad \forall h,$ la tâche A_h précéde la tâche B_{h+k}.
   Cette situation se rencontre dans le cas de productions industrielles en série.

3. Les tâches dans un atelier contenant m machines sont regroupées en *jobs* (travaux), chacune d'elles s'exécutant sur une machine différente (on aura donc m tâches dans un job).

   Un job peut contenir :

   - des tâches indépendantes, on parle alors de problème *open-shop*,
   - des tâches liées par un ordre total. Cet ordre n'étant pas forcément le même pour tous les jobs, on parle alors de problème *job-shop*.
   - Dans le cas où il y a le même ordre total sur tous les travaux, on parle de problème *flow-shop*.

8.1.3 Les ressources

Deux types de ressources seront considérés :

1. Les ressources *renouvelables* qui restent disponibles après avoir été allouées à une ou plusieurs tâches (par exemple les machines).

2. Les ressources *consommables* qui peuvent devenir non disponibles après avoir été allouées à une ou plusieurs tâches (par exemple les matières premières, l'argent, ...).

8.1.4 Les fonctions économiques

Les fonctions èconomiques ou *critères* à optimiser dans des problèmes d'ordonnancement dépendent en général des paramètres suivants :

1. Date de fin d'exécution d'une tâche.

2. Retard d'une tâche ou indicateur de retard (existence ou non-existence d'un retard).

Ils peuvent exprimer :

1. Une durée totale de l'ordonnancement,

2. Un respect des dattes au plus tard,

3. Une minimisation d'un coût,

4. Un nombre d'interruptions (préemption).

8.1.5 Notations

Dans la suite du cours, noous opterons pour les notations suivantes:

- I ensemble de tâches,
- $i, j, ... \in I$, tâches appartenant à I,
- p_i durée de la tâche i,
- r_i date de début au plus tôt d'exécution de la tâche i,
- d_i date de fin au plus tard d'exécution de i,
- t_i date de début d'exécution de i,
- c_i date de fin d'exécution de i,
- w_i poids associé à i,
- P_{ik} temps d'exécution de i sur la machine k.

Une contrainte potentielle exprimant une antériorité d'une tâche i par rapport à une tâche j s'exprimera par une inégalité. Par exemple l'inégalité $t_j - t_i \geq a_{ij}$ exprimera le fait que la tâche j débutera quand a_{ij} unités de temps se seront écoulées aprè le début d'exécution de la tâche i.

Exercice. Exprimer en fonction des données et des notations introduites précédemment

- la durée totale d'un ordonnancement,
- le retard d'une tâche i,
- l'indicateur de retard d'une tâche i.

8.2 Représentation des problèmes d'ordonnancement

8.2.1 Le diagramme de Gantt

Le diagramme de Gant est une méthode de représentation d'une solution à un problème d'ordonnancement. Cette méthode est ancienne, mais elle est encore utilisée dans de nombreux logiciels de gestion. C'est un outil de "visualisation" simple, son utilisation est possible dans des problèmes ne comportant pas de nombreuses tâches (risque d'illisibilité du diagramme).

Nous décrivons cette méthode à partir des exemples suivants:

exemple On considère un problème d'ordonnancement à cinq tâches.

- $I = \{1, 2, 3, 4, 5\}$ avec
- $p_1 = 6, p_2 = 3, p_3 = 4, p_4 = 5, p_5 = 5$,
- utilisant respectivement 4, 1, 3, 2, 3 unités de ressource 1 et
8, 7, 10, 10, 4 unités de ressource 2 ,
- lorsque les dates d'exécution respectives sont 0, 3, 6, 8, 10.

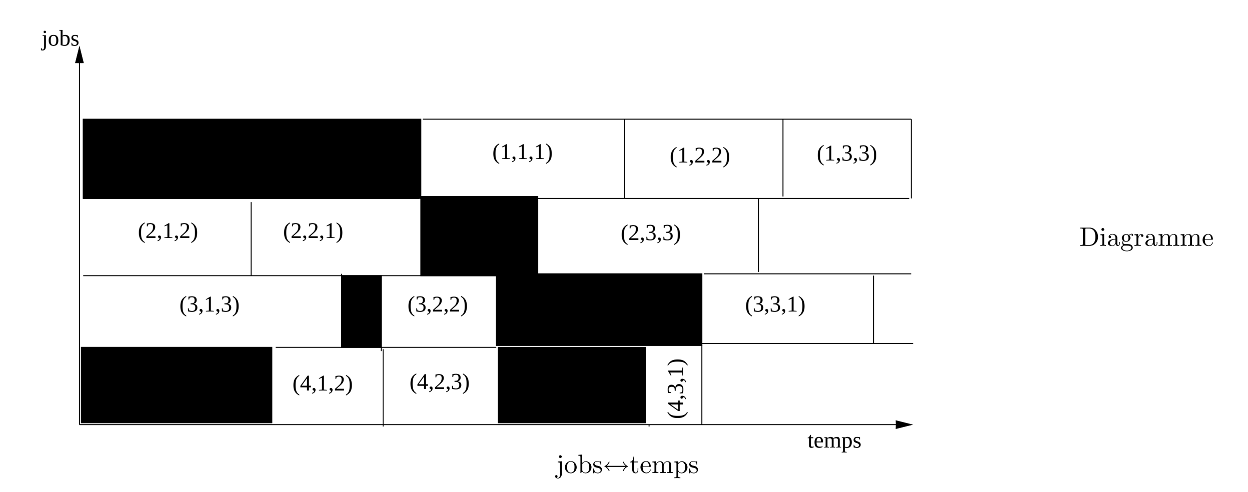
tâches↔temps

Diagramme

ressources↔temps

Diagramme

exemple Ordonnancement dans un atelier.
On a une représentation job par job. Dans notre exemple, on va considérer quatre travaux constitués de trois tâches. On utilisera la notation (i, j, k) pour représenter la $j^{ème}$ tâche du travail i sur la machine k.



jobs↔temps

Diagramme

8.2.2 Les graphes

Deux formulations sont possibles :

- potentiel-tâches
- potentiel-étapes (plusieurs représentations sont possibles dans ce cas).

Formulation potentiel↔tâches

On considèrera deux tâches i et j parmi n tâches, de durée respective d_i et d_j, et les dates de début t_i et t_j.

On traduira graphiquement et par des inégalités de type potentiel↔tâche un certain nombre de contraintes :

1. Postériorité au plus tôt.
   j commence au plus tôt un temps a_{ij} unités de temps après le début de i. L'inégalité correspondante à cette contrainte est :

   $t_j - t_i \geq a_{ij}$ représentée par l'arc $i. \xrightarrow{a_{ij}} .j$

2. Postériorité au plus tôt avec retard.
   j commence au plus tôt un temps Δ_{ij} unités de temps après l'achèvement de i. L'inégalité correspondante à cette contrainte est :

   $t_j - t_i \geq d_i + \Delta_{ij}$ représentée par l'arc $i. \xrightarrow{d_i+\Delta_{ij}} .j$

3. Postériorité au plus tôt avec chevauchement.
   j commence au plus tôt après une fraction c_{ij} de la durée de i. L'inégalité correspondante à cette contrainte est :

   $t_j - t_i \geq c_{ij}d_i$ représentée par l'arc $i. \xrightarrow{c_{ij}+d_i} .j$

4. Postériorité au plus tard.
   j commence au plus tard un temps v_{ji} unités de temps après le début de i. L'inégalité correspondant à cette contrainte est :

   $t_j \leq t_i + v_{ji}$ (ou $t_i - t_j \geq -v_{ji}$) représentée par l'arc $i. \xrightarrow{-v_{ji}} .j$

5. Encadrement du début de la tâche j.
   j commence au plus tôt un temps v_{ij} unités de temps après le début de i et au plus tard un temps v_{ji} unités de temps après le début de i. Les inégalités correspondant à cette contrainte sont :

   $$\begin{cases} t_j - t_i \geq v_{ij}. \\ t_j \leq t_i + v_{ji} \quad (ou\ t_i - t_j \geq -v_{ji}). \end{cases}$$ représentées par

   vij
   i j
   -v ij

6. Compatibilité de cadence.
   Lorsqu'une fraction c_{ji} de la durée de j s'est écoulée, au moins une fraction c_{ij} de la durée de i doit s'être écoulée ($0 < c_{ij}, c_{ji} < 1$). L'inégalité correspondante à cette contrainte est :

   $t_j + c_{ji}d_j \geq t_i + c_{ij}d_i$ (ou $t_j - t_i \geq c_{ij}d_i - c_{ji}d_j$) représentée par l'arc $i. \xrightarrow{c_{ij}d_i - c_{ji}d_j} .j$

Le graphe potentiel↔tâches

Un graphe potentiel↔tâches modélise un problème d'odonnancement avec contraintes potentielles. Les tâches considérées seront les n tâches du projet ainsi que deux tâches fictives "début" et "fin" de projet notées respectivement 0 et n+1. On associera :

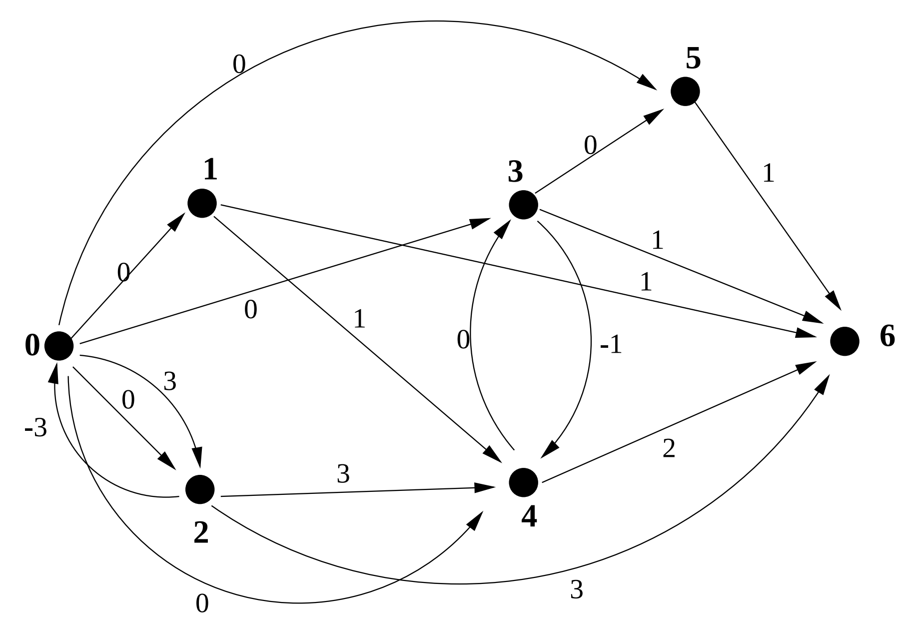
- les tâches aux sommets (le nombre de sommets est égal au nombre de tâches),
- les contraintes potentielles aux arcs tels que construits précédemment.

exemple Considérons l'ensemble de 5 tâches $\{ 1, 2, 3, 4, 5 \}$ soumises aux contraintes :

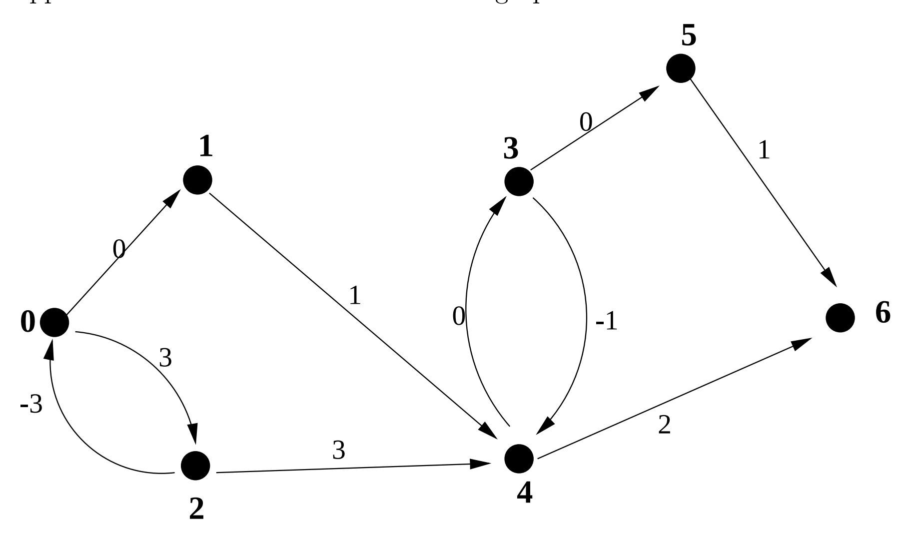
1. La tâche 2 commence à la date 3,
2. les tâches 3 et 4 doivent se chevaucher sur au moins une unité de temps,
3. la tâche 4 ne peut commencer qu'après la fin des tâches 1 et 2.
4. la tâche 5 ne peut commencer avant le début de la tâche 3. Les durées des tâches étant $d_1 = d_3 = d_5 = 1, d_2 = 3, d_4 = 2$, les inégalités respectives correspondant aux contraintes seront :

$$t_2 - t_0 \geq 3ett_0 - t_2 \geq -3 \text{ (encadrement du début de 2)}$$
$$t_3 \leq t_4 + d_4 - 1 \text{ et } t_4 \leq t_3 + d_3 - 1$$
$$t_4 - t_2 \geq d_2 \text{ et } t_4 - t_1 \geq d_1$$
$$t_5 \geq t_3.$$

On aura le graphe potentiel↔tâches suivant :



Après suppression des arcs inutiles on obtient le graphe :



L'arc 1. ⟶ .6 est inutile, car l'arc 1. ⟶ .4 indiquant que 4 est exécutée après est suffisant.

Remarque Dans un graphe potentiel↔tâches, une contrainte disjonctive entre deux tâches i, j pourra être représentée comme suit : $i. \longrightarrow\longleftarrow .j$

Formulation potentiel↔étapes

La formulation potentiel↔étapes est la première représentation d'un ordonnancement a être utilisée historiquement (1958). Cette formulation appelée aussi P.E.R.T. (*Program Evaluation and Research Technic*) construit un graphe potentiel dont les sommets sont associés à des événements du type "fin de tâche", "début de tâche" et dont les arcs sont associés aux tâches.

Comme pour tout ce qui précède, la méthode PERT nésessite tout d'abord de définir :

- le projet à réaliser,
- les différentes tâches,
- leur durée,
- les liens qu'il y a entre elles.

On représentera les étapes par des cercles (sommets du graphe) et les tâches par des flèches (arcs du graphe).

exemple Le graphe
A B C D E
15 20 12 7 1

représente le problème d'ordonnancemenmt résumé dans le tableau de données

Tâches	Durée en unité de temps	Tâches antérieures
A	15	
B	20	A
C	12	B
D	7	C
E	1	D

Principes de la représentation On utilise la colonne "tâches antérieures",

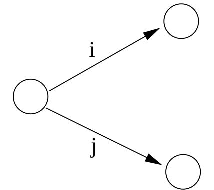
- on représente d'abord les tâches sans antécédent à un premier niveau dans le graphe (dans l'exemple A est à un premier niveau),
- on supprime de la colonne les tâches représentées, puis on représente les tâches qui n'ont plus d'antécédent à un second niveau (dans l'exemple B est à un second niveau),
- ...

On peut ainsi obtenir plusieurs niveaux dans un graphe.
Les principes de base permettant la construction d'un graphe PERT sont les suivants :

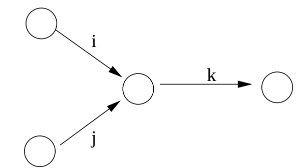
1. un PERT a un seul sommet "Début" et un seul sommet "Fin".
2. 2 tâches i et j se succédant sont représentées par



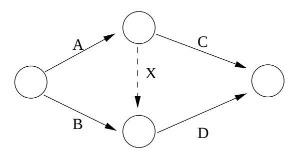
3. 2 tâches i et j se simultanées sont représentées par



4. 2 tâches i et j précédant une tâche k sont représentées par



5. pour les besoins de représentation, des tâches fictives sont parfois nécessaires, on les représente par des flèches en trait discontinu, par exemple



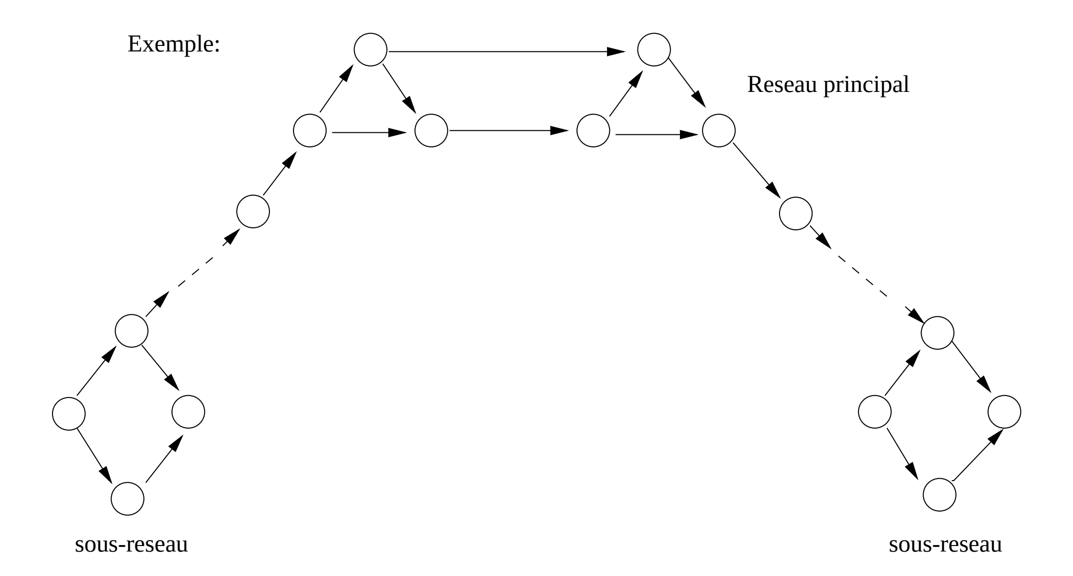
X est une tâche fictive introduite pour représenter le fait que A précède D.

Exercice Représenter en PERT les différentes contraintes potentielles des pages précédentes.

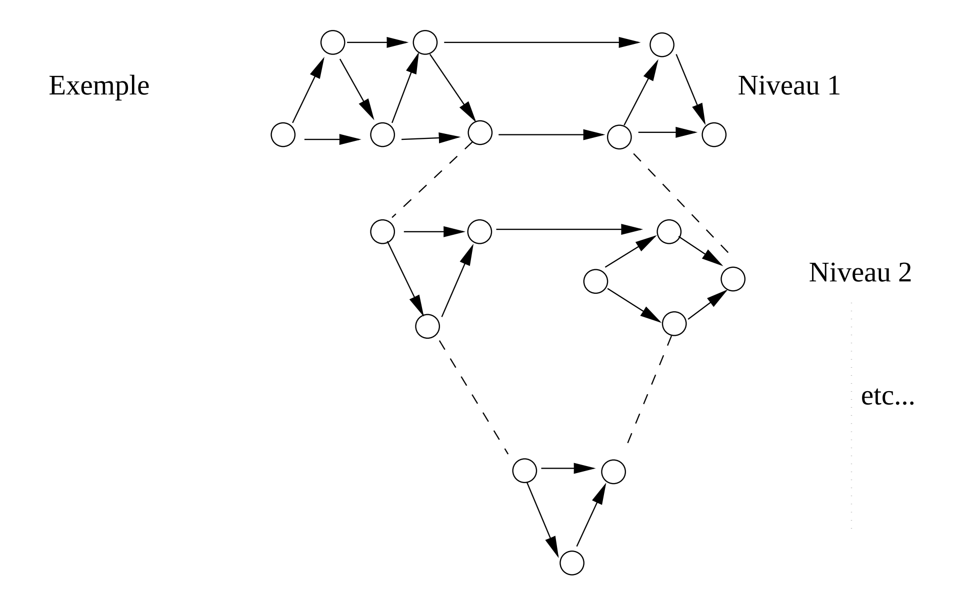
Le multi-PERT (ou réseau PERT)

Quand un graphe PERT devient trop complexe, on peut le diviser soit en sections, soit en niveaux.

• Division en sections



• Division en niveaux



8.3 Méthode du chemin critique

Les chemins de valeur maximale, joignant l'entrée et la sortie d'un graphe potentiel jouent un rôle important dans un problème d'ordonnancement. Toute tâche appartenant à ces chemins ne peuvent être retardée sans que le retard soit répercuté sur la durée de l'ordonnancement. De telles tâches sont dites *critiques* et les chemins auquels elles appartiennent son appelés *chemins critiques.*

La méthode du chemin critique permet de détecter les tâches critiques.

8.3.1 Description de la méthode

Un graphe potentiel nous permet de mesurer les quantités suivantes (pour certaines à l'aide des algorithmes de Ford ou de Dijsktra) :

1. *La date au plus tôt*, de début d'une tâche i que l'on notera t_i. Cette date est égale à la longueur du plus long chemin de l'entrée 0 au sommet i du graphe. On a :

$$t_i = \max_{j \in \Gamma_i^{-1}} (t_j + d_j)$$

Γ_i^{-1} étant l'ensemble des prédécesseurs de i dans le graphe, et d_j étant la durée d'une tâche j.

2. *La durée minimale du projet* t_{n+1}, est la longueur du plus long chemin de 0 à $n+1$.

3. *La date au plus tard* T_i du début de la tâche i. On a $T_i = t_{n+1} - l_{i\ n+1}$ où $l_{i\ n+1}$ est la longueur du plus long chemin de i à $n+1$.

4. *La marge* m_i de la tâche i est la différence $T_i - t_i$.

Les tâches critiques seront les tâches pour lesquelles la marge est nulle.

On rappelle que pour qu'une tâche puisse commencer, il est nécessaire que toutes les tâches qui la relient à la tâche 0 soient réalisées.

On recherchera alors un ordonnancement qui minimise la durée totale et d'autre part on identifiera les tâches critiques.

Algorithme de recherche des dates au plus tôt.

1- poser $t_0 = 0$

2- prendre les sommets j par niveau croissant dans le graphe potentiel et faire

$$t_j = \max_{k \in \Gamma_j^{-1}} (t_k + d_k)$$

Algorithme de recherche des dates au plus tôt.

1- poser $T_{n+1} = t_{n+1}$

2- prendre les sommets par niveau décroissant dans le graphe potentiel et faire

$$t_j = \min_{k \in \Gamma_j} (T_k) - d_j$$

La détermination des marges et des tâches critiques se fait alors de façon évidente.

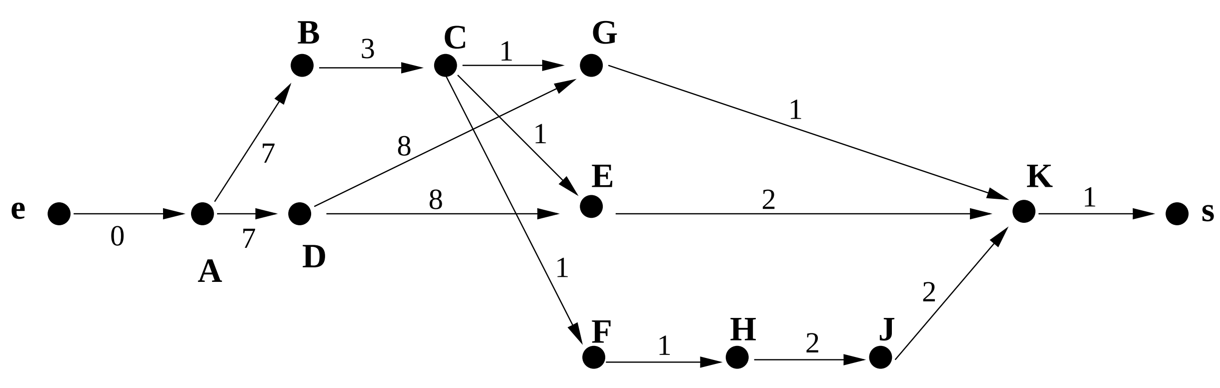
exemple La construction d'un ouvrage nécessite la réalisation d'un certain nombre de tâches que l'on résume dans le tableau de données suivant :

Code tâches	Description	Durée	Tâches antérieures
A	Travaux de maçonnerie	7	
B	Charpente et toiture	3	A
C	toiture	1	B
D	Sanitaire + électricité	8	A
E	Façade	2	D, C
F	Fenêtres	1	D, C
G	Jardin	1	D, C
H	Plafonnage	3	F
J	Peinture	2	H
K	Emménagement	1	E, G, J

1. Donner le graphe potentiel↔tâches modélisant ce problème.

2. Donner les dates au plus tôt de début de chaque tâche, la durée minimale du projet, les dates au plus tard de début de chaque tâche, la marge de chaque tâche, les tâches critiques.

- Graphe potentiel↔tâches :



- Dates de début au plus tôt :
  $t_e = 0, t_A = 0, t_B = 7, t_D = 7, t_C = 10, t_G = 15, t_E = 15, t_F = 11, t_H = 12, t_J = 14, t_K = 17, t_s = 18$ (durée minimale du projet).

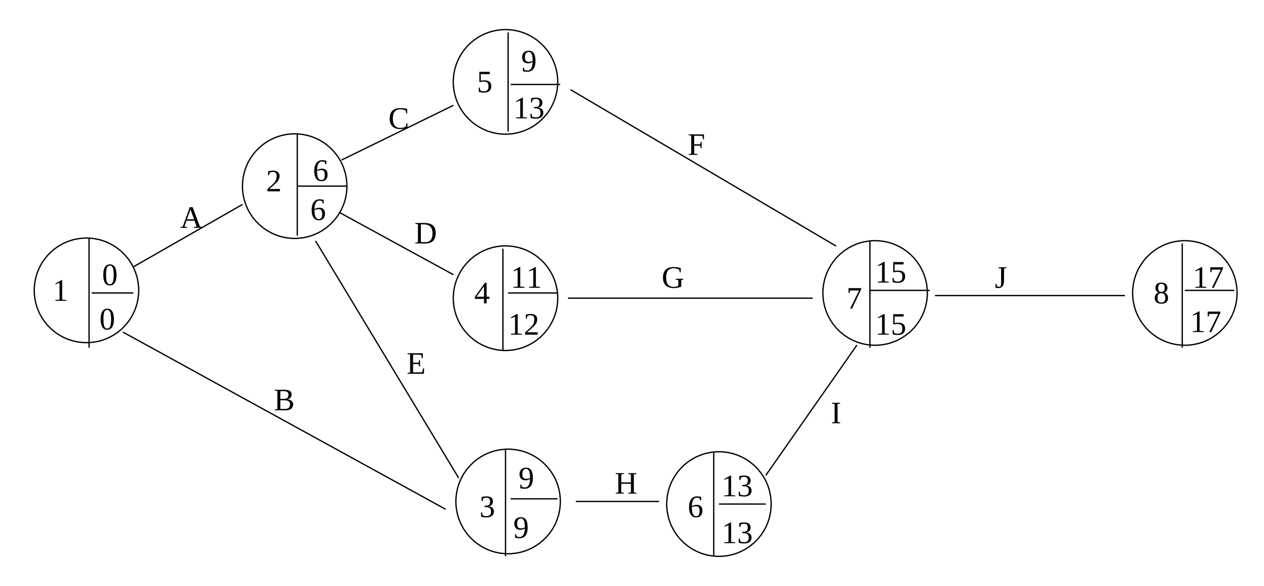
- Dates de début au plus tard :
  $T_s = 18, T_K = 17, T_J = 15, T_H = 13, T_F = 12, T_E = 15, T_G = 16, T_C = 11, T_D = 7, T_B = 8, T_A = 0, T_e = 0.$

- Marges :
  $m_A = 0, m_B = 1, m_C = 1.m_D = 0, m_E = 0, m_F = 1, m_G = 1, m_H = 1, m_J = 1, m_K = 0.$

- Tâches critiques :
  A, D, E, K.

exemple Dans ce second exemple nous utilisons une modélisation à l'aide d'un graphe potentiel↔étapes. Le tableau de données qui suit correspond aux différentes tâches d'un projet de réalisation d'un nouveau produit dans une entreprise.

Code tâches	Description	Durée	Tâches antérieures
A	Avant-projet	6	
B	Etude de marchés	2	
C	Etude de faisabilité	3	A
D	Réalisation	5	A
E	Définition d'une politique publique	3	A
F	Estimation des coûts	2	C
G	Présentation des prototypes	3	D
H	Détermination du prix	4	B, E
I	Evaluation du chiffre d'affaire	2	H
J	Synthèse avant lancement d'une série	2	F, G, I



Dans cette représentation schématique, les évènements (fin et début de tâches) apparaissent en tant que cercles divisés en trois parties :

1. la partie gauche porte le numéro de sommet,
2. la partie supérieure droite porte la valeur de début au plus tôt de la tâche qui a pour origine ce sommet,
3. la partie inférieure droite porte la valeur de début au plus tard de la tâche qui a pour origine ce sommet.

Cette représentation nous permet de lire rapidement sur le graphe quelles sont les tâches critiques. Dans notre cas A, C, D, E H, I, J sont critiques.

Les marges pour les tâches F et G étant respectivement de 4 et de 3 unités de temps. La durée du projet est de 17 unités de temps.

8.4 Méthodes sérielles

Une des idées les plus simples pour construire une solution d'un problème d'ordonnancement consisterait à construire une liste de tâches ordonnées par ordre de priorité d'exécution (cet ordre pouvant être statique ou dynamique). Les méthodes sérielles utilisent cette idée en tenant compte de la présence de ressources renouvellables.

" *A l'instatnt t, on affecte, parmi les tâches prêtes, c'est à dire celles dont tous les prédecesseurs sont achevées et qui utilisent moins de ressource que la quantité disponible à t, la tâche de plus haute priorité.*"
Les méthodes sérielles permettent la construction d'une solution approchée avec certains inconvénients. En effet l'exécution d'une tâche prête (répondant à la contrainte de disponibilité de ressource) s'il en existe peut être moins intéressante que le fait de laisser momentanément des ressources inutilisées en vue d'exécuter une autre tâche.
Dans l'établissement des priorités, en général, deux règles sont très utilisées:
. Ordonner suivant les dates au plus tard;
. Ordonner suivant les dates de fin au plus tard.

8.4.1 Algorithme de liste.

initialisation.
$U = \emptyset,\ t = 0$ /* U est l'ensemble des tâches ordonnancées. */
Etape courante.
tant que $U \neq I$
Si (le sous-ensemble U' de complémentaire de U dans I des tâches disponibles à t est non vide)
{
Prendre la tâche i de plus grande priorité;
$t_i = t$;
$U = U \cup \{i\}$
}
Sinon
{
Déterminer le plus petit instant t où une tâche dans complémetaire de U dans I devient disponible.
}

8.4.2 Exemple.

On considère un problème d'ordonnancement dont les données sont résumées dans le tableau suivant:

	a	b	c	d	e	f	g	h	i	j	dispo.
Resource 1	3	3	1	1	1	2	3	2	1	2	5
Ressource 2	0	0	0	1	1	1	0	1	0	0	1
Date au plus tôt	0	2	0	10	6	2	18	12	27	7	
Date au plus tard	0	6	18	10	12	20	18	25	27	30	
Durée	10	4	2	8	6	5	9	2	7	4	
Tâches antérieures	/	/	/	a, b	b	c	d, e	e, f	g, h	f	

1- Tracer le graphe potentiel↔tâches associé.
2- Donner les traces d'exécution de l'algorithme.
3- Tracer les diagrammes de Gantt correspondant.

Appendix A

Le Langage algorithmique C', Manuel d'utilisation.

Le langage algorithmique que nous proposons devra permettre

1. Une meilleure écriture du cours. En effet en plus de la définition d'une syntaxe pour l'écriture des programmes, notamment dans la clause ALGOs d'un schéma abstrait de résolution, il faudra donner un certain nombre de règles facilitant la lisibilité et la cohérence du contenu des différents chapitres ainsi qu'une meilleure compréhension des aspects théoriques liés à l'algorithmique. L'utilisation directe, dans les programmes en C', de procédures ou de types définis abstraitement étant possible.
2. Une présentation naturelle et claire des différents programmes du cours en réduisant au minimum les contraintes de syntaxe, de façon à ce que la structuration des programmes soit perçue comme une question devant se poser avant celle du choix d'un langage de programmation.
3. Un passage simplifié vers des programmes en langage de haut niveau notamment en C pour la programmation impérative et en smalltalk pour la programmation objet. Un tel passage devant être, sinon automatique, du moins largement facilité pour réduire au maximum la phase finale de construction de programmes.

Le langage proposé sera aussi proche que possible d'un langage naturel, et en l'occurence pour nous du Français.

A.1 Les structures de contrôle.

• **La séquence directe.**

```
{
 instruction1          /* Commentaires */
 instruction2
   ⋮
 instructionn
}
```

Chaque instruction peut être suivie d'un commentaire facilitant la compréhension du programme (ces commentaires sont encadrés par /* et */.
L'accolade ouvrante indique le début de la séquence et la fermante, sa fin.

• **Les structures conditionnelles.**

```
si condition                 si condition
    séquence                     séquence1
                              sinon
                                 séquence2
```

• **Les structures itératives.**

1- **tant que** condition **faire**

séquence séquence
tant que condition

Il est important d'insister sur le fait que dans toutes les structures précédentes, les expressions conditionnant l'exécution de telle ou telle partie d'un programme devront être des expressions logiques dont l'évaluation pourra se faire simplement.

2- **pour** i=init **à** term
séquence

Un inconvénient dans l'utilisation de la structure **pour** est du au fait que l'itération n'est pas modifiable et est fixée à un élément par itération.

Pour forcer la sortie dune boucle, on utilisera l'instruction **sortir-boucle**.

A.2 Les fonctions et procédures.

Un appel à une fonction ou procédure définie de façon abstraite ou construite se fera par son nom souligné ou son nom souligné et indexé par p lorsqu'il s'agit d'une primitive d'un TAD. Le nom est suivi de la liste des données d'entrée nécessaires.
L'écriture d'un programme correspondant à une fonction ou procédure se fait suivant le schéma:
(*type* des donnees retournées)nom de la procédure(liste des *types* et noms des arguments)
{
déclarations des types et noms des variables locales
séquence directe
}
Une instruction particulière pourra être associée à une procédure de la manière suivante:
retourner(donnée) dans le cas où la procédure retourne une donnée.
ou
retourner si elle ne retourne rien.

A.3 Types et opérateurs.

Les types sont notés en italique. La déclaration d'une variable ou d'un paramètre et de son type se fait suivant le schéma:
type nom-variable.

A.3.1 Les types primitifs.

Les types primitifs autorisés sont: *entier, réel, car, booléen.*
Les variables de boucle et leur type peuvent ne pas être déclarées s'il n'y a pas d'ambiguïté possible.

A.3.2 Les types abstraits.

S'il est défini abstraitement, un type abstrait peut être soit directement utilisé dans un programme, soit représenté par un type construit. Cette seconde représentation est alors à un niveau d'abstraction inférieur.
On appellera "type construit" tout type ne faisant pas partie des types primitifs. On a les schémas syntaxiques:
type *nouveau-type* = *ancien-type*
ou encore:
type *nouveau-type* =
{
ancien-type1 nom-champ1
ancien-type2 nom-champ2

⋮
ancien-typen nom-champn
}

Dans ce cas si une donnée de nom X est de type *nouveau-type*, l'accès à un champ se fait de la manière suivante:

X.nom-champ

On définit les règles suivantes sur les types abstraits:
- La valeur d'une variable déclarée sur un type abstrait est une référence à une donnée de ce type.
- On ne peut accéder aux données d'un type abstrait que par les primitives apparaissant dans les spécification du type ou par des fonctions et procédures définies à partir de ces primitives.
- Une variable ne faisant référence à aucune donnée vaut NIL (qui est la valeur FAUX pour une référence).
- Il est possible de déclarer à la fois le type des données de même nature faisant partie d'un groupe et le type du groupe. On utlise alors le schéma:

type-groupe[type-donnees]

A.3.3 Les tableaux.

Le TAD *tableau* ayant une importance particulière par la suite, nous l'autorisons en tant que type prédéfini dans le langage avec le schéma:

nom-tableau[type-élément][dimension]

A.3.4 Les opérateurs.

- Les opérateurs de comparaison sont: $=, \neq, <, >, \leq, \geq$.

- Les opérateurs Booléens sont: ET, OU, NON. Les constantes Booléennes sont: VRAi resp. FAUX ou encore ce qui est équivallent: "valeur numérique d sont: VRAi resp. FAUX ou encore ce qui est équivallent: "valeur numérique différente de zéro" resp. "valeur numérique égale à zéro".

- Les opérateurs arithmétiques sont: $+, -, \times, /$ ainsi que modulo.

A.3.5 Les instructions primitives.

. L'affectation d'une valeur d'un type donné à une variable compatible avec ce type se fait comme suit:

nom-variable == valeur

. Les instructions d'écriture et de lecture se font à l'aide des fonctions prédéfinies **écrire** et **lire**. Un message que l'on veut écrire appaîtra entre deux quotes en tant qu'argument de **écrire**. L'absence des quotes sous-entent que ce message a une valeur et c'est alors cette valeur qui est écrite.